ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

В.В. Онищенко, Т.П. Довженко

СПЕЦІАЛІЗОВАНІ МОВИ ПРОГРАМУВАННЯ

Київ-2019

АНОТАЦІЯ

Спеціалізовані мови програмування. 1-е вид. - К.: Державний університет телекомунікацій, 2019. - 146с.

У посібнику “Спеціалізовані мови програмування” розглянуто основи мови програмування Python 3 версії: основні алгоритмічні структури, типи даних, розкрито поняття модуля. Також показано функціональний підхід для розв’язання прикладних задач з програмування, наведені приклади роботи з математичними бібліотеками, різними форматами даних (XML, CSV, JSON), проведено огляд мережевих та web-додатків на мові Python.

Для викладачів та студентів IT ВУЗів та тих, хто цікавиться програмуванням на мові Python 3.

Рецензенти: ____
(прізвище, ініціали, вчений ступінь, звання)

Рекомендовано (ким, чим)____
(назва, підрозділ, відомство, місто)

ЗМІСТ

ВСТУП 7

1. **ВВЕДЕННЯ В МОВУ ПРОГРАМУВАННЯ PYTHON** 8

1.1. Що таке Python? 8

1.2. Як описати мову? 8

1.3. Історія мови Python 9

1.4. Основні алгоритмічні конструкції 10

1.4.1. Послідовність операторів 10

1.4.2. Умовні оператори 10

1.4.3. Цикли 11

1.4.4. Функції 13

1.4.5. Винятки 13

1.5. Вбудовані типи даних 14

1.5.1. Числа 16

1.5.2. Тип bool 18

1.5.3. Тип string та тип unicode 19

1.5.4. Тип tuple 20

1.5.5. Тип list 20

1.5.6. Робота з послідовностями 21

1.5.7. Тип dict 22

1.5.8. Тип file 22

1.6. Вирази 23

1.7. Імена 24

Практична робота 26

Висновок 27

Питання для самоконтролю 27

2. **ОСНОВНІ СТАНДАРТНІ МОДУЛІ PYTHON** **28**

2.1. Поняття модуля 28

2.2. Модулі в python 29

2.2.1. Шлях пошуку модулів 30

2.2.2. Компільовані файли 31

2.2.3. Стандартні модулі 31
2.3. Огляд стандартної бібліотеки 32
2.3.1. Інтерфейс операційної системи 32
2.3.2. Шаблони розширення файлових назв 33
2.3.3. Аргументи командного рядка 33
2.3.4. Переспрямування виводу помилок та вихід із програми 34
2.3.5. Пошук за шаблоном 34
2.3.6. Математика 34
2.3.7. Доступ до мережі Інтернет 35
2.3.8. Час і число 35
2.3.9. Ущільнення даних 36
2.3.10. Обчислення продуктивності 36
2.3.11. Контроль якості 37
2.3.12. Форматування виводу 38
2.3.13. Шаблони 39
Практична робота 40
Висновок 42
Питання для самоконтролю 42
3. **ЕЛЕМЕНТИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ** **43**
3.1. Що таке функціональне програмування? 43
3.2. Функціональна програма 43
3.3. Функція: визначення і виклик 44
3.3.1. Стандартні значення аргументів 46
3.3.2. Ключові аргументи 48
3.3.3. Списки аргументів довільної довжини 50
3.4. Рекурсія 51
Практична робота 55
Висновок 57
Питання для самоконтролю 57
4. **ОБЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ** **58**
4.1. Основні поняття 58

4.2. Абстракція і декомпозиція 60
4.3. Об'єкти 61
4.4. Типи і класи 62
4.5. Успадкування, інкапсуляція, поліморфізм 65
Практична робота 66
Висновок 71
Питання для самоконтролю 71
5. **ЧИСЕЛЬНІ АЛГОРИТМИ. МАТРИЧНІ ОБЧИСЛЕННЯ** **72**
5.1. Модуль Numpy 72
5.2. Створення масиву 72
5.3. Функції для роботи з масивами 76
5.4. Індекси, зрізи, ітерації 78
5.5. Функції модуля Numpy 81
5.5.1. Маніпуляції з формою 81
5.5.2. Об'єднання масивів 82
5.5.3. Розбиття масиву 83
Практична робота 84
Висновок 86
Питання для самоконтролю 86
6. **ОБРОБКА ТЕКСТІВ** **87**
6.1. Рядки 87
6.2. Кодування python-програми 87
6.3. Рядкові літерали 88
6.4. Операції над рядками 89
6.5. Модуль UNICOD 89
6.6. Методи рядків 91
Практична робота 93
Висновок 97
Питання для самоконтролю 97
7. **РОБОТА З ДАНИМИ В РІЗНИХ ФОРМАТАХ** **98**
7.1. XML 98

7.1.1. Формування XML-документа 99
7.1.2. Аналіз XML-документа 101
7.2. Формат CSV 104
7.3. Формат JSON 105
7.3.1. Основні методи 106
7.3.2. Класи модуля JSON 108
Практична робота 109
Висновок 111
Питання для самоконтролю 111
8. **РОЗРОБКА WEB-ДОДАТКІВ** **112**
8.1. CGI додатки 112
8.1.1. Модуль cgi 114
8.1.2. Модуль cgitb 114
8.2. Стандарт WSGI 117
8.3. Введеня у фреймворк Django 118
Практична робота 120
Висновок 127
Питання для самоконтролю 127
9. **МЕРЕЖНІ ДОДАТКИ НА PYTHON** **128**
9.1. Робота із сокетами 128
9.2. Модуль smtplib 131
9.3. Модуль poplib 133
9.4. Модулі для клієнта www 137
9.4.1. Функції для завантаження мережевих об'єктів 137
9.5. XML-RPC-сервер 139
Практична робота 140
Висновок 143
Питання для самоконтролю 143
ВИСНОВКИ **144**
СПИСОК ЛІТЕРАТУРИ **146**

ВСТУП

У даному методичному посібнику розкрито основні теми, які потрібні для роботи на мові програмування Python 3. Python - стабільна та поширена мова. Вона використовується в багатьох проектах та в різних якостях: як основна мова програмування або для створення розширень та інтеграції додатків. На Python реалізована велика кількість проектів, також вона активно використовується для створення прототипів майбутніх програм. Тому вивчення даної мови програмування є перспективним.

У посібнику розглянуто основні типи та структури даних мови Python 3. На прикладах показані можливості основних бібліотек мови Python. Окремо проведено огляд бібліотеки Numpy для проведення високоточних математичних обчислень. Наведені приклади реалізації прикладних задач мовою Python 3 на основі функціональної та об’єктно-орієнтованої парадигм програмування. Показані приклади роботи з найпоширенішими структурами даних (XML, CSV, JSON). Також було проведено огляд web-додатків та бібліотек роботи з мережевими технологіями.

З огляду на світові тенденції, які склалися в програмуванні в цілому, розглянутий матеріал в даному посібники має актуальне значення в контексті вивчення мови прогамування Python.

1. ВВЕДЕННЯ В МОВУ ПРОГРАМУВАННЯ PYTHON

1.1. Що таке Python?

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня з строгою динамічною типізацією. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднання існуючих компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій так і у вихідній формі на всіх основних платформах. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

В стандартний пакет Python входить велика стандартна бібліотека для вирішення широкого кола завдань. В Інтернеті доступні якісні бібліотеки для Python по різних предметних областях: засоби обробки текстів і технології Інтернет, обробка зображень, інструменти для створення додатків, механізми доступу до баз даних, пакети для наукових обчислень, бібліотеки побудови графічного інтерфейсу і т.д.

1.2. Як описати мову?

Найбільш повний опис мови Python можна знайти за посиланням https://www.python.org/ .

В даному методичному посібнику пропонується розглянути мову одночасно в декількох аспектах, що досягається набором прикладів, які дозволять швидше долучитися до реального програмування, ніж в разі звичайного академічного підходу.

Однак варто звернути увагу на правильний підхід до опису мови. Створення програми - це завжди комунікація, в якій програміст передає комп'ютеру інформацію, необхідну для виконання останнім дій. Те, як ці дії розуміє програміст

(тобто "сенс"), можна назвати *семантикою*. Засобом передачі цього сенсу є *синтаксис* мови програмування. Ну а те, що робить інтерпретатор на підставі переданого, зазвичай називають *прагматикою*. При написанні програми дуже важливо, щоб в цьому ланцюжку не виникало збоїв.

Синтаксис – повністю формалізована частина: його можна описати на формальній мові синтаксичних діаграм. Виявом прагматики є сам інтерпретатор мови. Саме він читає записаний відповідно до синтаксису код" і перетворює його в дію відповідно до закладеномого в ньому алгоритму. Неформальним компонентом залишається тільки семантика. Синтаксис мови Python має потужні засоби, які допомагають наблизити розуміння проблеми програмістом до її "розуміння" інтерпретатором.

1.3. Історія мови python.

Python – порівняно «молода» мова. Створюючи її у 1990-1991 роках, її автор Гвідо ван Россум (Guido van Rossum) врахував усі переваги та недоліки попередніх мов програмування. Мова почала вільно поширюватися через Інтернет і сподобалася іншим програмістам. З 1991 року Python є цілком об'єктно-орієнтованим. Python також запозичив багато рис таких мов, як C, C++ й окремі риси функціонального програмування. 3 грудня 2008 року, після тривалого тестування, вийшла перша версія Python 3 (або Python 3.0, також використовується скорочена Py3k). У Python 3 усунено багато недоліків архітектури з максимально можливим (але не повним) збереженням сумісності зі старішими версіями. На сьогодні підтримуються обидві гілки розвитку (Python 3.6 і 2.7).

У наш час Python має широку область застосування. Це – мова, яка розвивається, її використовують в реальних проектах. Засоби для роботи з Python відносяться до категорії вільно поширюваного програмного забезпечення, що гарантує відсутність претензій щодо використання «інтелектуальної власності». Існує множина засобів, які полегшують процес створення програм на Python, серед них можна виділити спеціалізовані лексичні аналізатори і редактори для програмістів (наприклад, Kate і Bluefish), інтегровані середовища розробки. Багато

засобів для роботи з Пітон є кросплатформними, в конструкціях мови підтримується багатобайтове кодування (Unicode), тому програми на Python легко переносити з одного середовища функціонування на інше.

1.4. Основні алгоритмічні конструкції.

Передбачається, що слухачі вже вміють програмувати хоча б на рівнішкільної програми, і тому цілком достатньо провести паралелі міжалгоритмічними конструкціями і синтаксисом Python.

1.4.1. Послідовність операторів

Послідовні дії описуються послідовними рядками програми. Варто, щоправда, додати, що в програмах важливі відступи, тому всі оператори, що входять до послідовність дій, повинні мати один і той самий відступ:

```
>>> a = 1
>>> b = 2
>>> a = a + b
>>> b = a - b
>>> a = a - b
>>> print(a, b)
2 1
```

При роботі з Python в інтерактивному режимі як би вводиться одна велика програма, що складається з послідовних дій. В наведеному вище прикладі використані оператори присвоювання і оператор *print*.

1.4.2. Умовні оператори

```
if a > b:
    c = a
else:
        c = b
```

Цю частину коду на Python інтуїтивно зрозумілий кожному, хто пам'ятає, що *if* по англійськи означає "якщо", а *else* - "інакше". Оператор розгалуження має в даному випадку дві частини, оператори кожної з яких записуються з відступом

вправо щодо оператора розгалуження. Більш загальний випадок - оператор вибору - можна записати за допомогою наступного синтаксису:

```
if a < 0:
   s = -1
elif a == 0:
   s = 0
else:
   s = 1
```

Варто зауважити, що elif - це скорочений else if. Без скорочення довелося б застосовувати вкладений оператор розгалуження:

```
if a < 0:
   s = -1
else:
   if a == 0:
      s = 0
   else:
      s = 1
```

На відміну від оператора print, оператор if-else - складений оператор.

1.4.3. Цикли

Третьою необхідною алгоритмічною конструкцією є цикл. За допомогою циклу можна описати повторювані дії. В Python є два види циклів: цикл ПОКИ (виконується деяка умова) і цикл ДЛЯ (всіх значень послідовності). Наступний приклад ілюструє цикл ПОКИ на Python:

```
s = "abcdefghijklmnop"
while s != "":
   print(s)
   s = s[1:-1]
```

Оператор while говорить інтерпретатору Python: "поки умова циклу вірна, виконуй тіло циклу". У мові Python тіло циклу виділяється відступом. Кожне виконання тіла циклу називається ітерацією. У наведеному прикладі забирається перший і останній символ рядка до тих пір, поки не залишиться порожній рядок.

Для більшої гнучкості при організації циклів застосовуються оператори *break* (перервати) і *continue* (продовжити). Перший скасовує виконання циклу, а другий -

продовжує цикл, перейшовши до наступної ітерації (якщо, звичайно, виконується умова циклу).

Наступний приклад читає рядки з файлу і виводить ті, у яких довжина більше 5 символів:

```
f = open("file.txt", "r")
while true:
   l = f.readline()
   if not l:
      break
   if len(l) > 5:
      print(l)
f.close()
```

У цьому прикладі нескінченний цикл переривається тільки при отриманні з файлу порожнього рядка (l <=5), що позначає кінець файлу.

У мові Python логічне значення несе кожен об'єкт: нулі, порожні рядки і послідовності, спеціальний об'єкт None і логічний літерал False мають значення "неправда", а інші об'єкти значення "істина". Для позначення істини зазвичай використовується 1 або True.

Цикл ДЛЯ виконує тіло циклу для кожного елемента послідовності. В наступному прикладі виводиться таблиця множення:

```
for i in range(1, 10):
   for j in range(1, 10):
      print("%2i" % (i*j))
   print()
```

Тут цикли *for* є вкладеними. Функція *range()* породжує список цілих чисел з напіввідкритого інтервалу [1, 10). Перед кожною ітерацієюлічильник циклу отримує чергове значення з цього списку. Напіввідкриті діапазони загальноприйняті в Python. Вважається, що їх використання більш зручно і викликає менше програмістів помилок. Наприклад, *range(len(s))* породжує список індексів для списку s (в Python-послідовності перший елемент має індекс 0). Для красивого виведення таблиці множення застосована операція форматування *%* (для цілих

чисел той же символ використовується для позначення операції взяття залишку від ділення).

1.4.4. Функції

Програміст може визначати власні функції двома способами: за допомогою оператора *def* або прямо в виразі, за допомогою *lambda*. Наведемо приклад визначення і виклику функції:

```
def price(hrn, kop=0):
   return "%i hrn. %i kop." % (hrn, kop)

print(price(8, 50))
print(price(7))
print(price(hrn=23, kop=70))
```

У цьому прикладі визначена функція двох аргументів (у тому числі другий має значення за замовчуванням - 0). Варіантів виклику цієї функції з конкретними параметрами також кілька. Варто лише зауважити, що при виконанні функції спочатку повинні йти позиційні параметри, а потім, іменовані. Аргументи зі значеннями за замовчуванням повинні слідувати після звичайних аргументів. Оператор ***return*** повертає значення функції. З функції можна повернути тільки один об'єкт, але він може бути кортежем з кількох об'єктів.

1.4.5. Винятки

У сучасних програмах передача управління відбувається не завжди так гладко, як в описаних вище конструкціях. Для обробки особливих ситуацій (таких як ділення на нуль або спроба зчитати неіснуючий файл) застосовується механізм винятків. Найкраще пояснити синтаксис оператора *try-except* можна наступним прикладом:

```
try:
   res = int(open('a.txt').read()) / int(open('c.txt').read())
   print(res)
except IOError:
```

```
    print("Input/outpet error")
except ZeroDivisionError:
    print("Zero division Error")
except KeyboardInterrupt:
    print("Keyboard Interrupt")
except:
        print("Error")
```

У цьому прикладі числа зчитуються з двох файлів і діляться один на одне. Внаслідок цих дій може виникнути кілька виняткових ситуацій, деякі з них відзначені в *except* (тут використані стандартні вбудовані виключення Python). Останній *except* в цьому прикладі вловлює всі інші винятки, що не були спіймані вище. Наприклад, якщо хоча б в одному з файлів знаходиться нечислове значення, функція *int()* порушить виняток *ValueError*, який зможе відловити останній *except*. Виконання частини *try,* в разі виникнення помилки, після виконання однієї з частин *except* вже не здійснюється

Винятки можна порушувати і з програми. Для цього служить оператор *raise*:

```
class MyError(Exception):
    pass
try:
    raise MyError("my error 1")
except MyError:
    print("Error:")
        raise
```

Для тверджень застосовується спеціальний оператор *assert*. він призводить до виключення типу *AssertionError*, якщо задана в ньому умова невірна. Цей оператор використовують для самоперевірки програми:

```
a = 1
b = 9
c = a + b
assert c == a + b
```

Крім описаної форми оператора, є ще форма try-finally для гарантованого виконання деяких дій при передачі управління зсередини оператора try-finally зовні. Він може застосовуватися для звільнення зайнятих ресурсів, що вимагає обов'язкового виконання, незалежно від того, що сталися всередині:

```
try:
...
finally:
    print("The programm has been done")
```

1.5. Вбудовані типи даних.

Всі дані в Python представлені об'єктами. Імена є лише посиланнями на ці об'єкти і не несуть навантаження по декларації типу. Значення вбудованих типів мають спеціальну підтримку в синтаксисі мови: можна записати літерал рядка, числа, списку, кортежу, словника. Синтаксичну ж підтримку операцій над вбудованими типами можна легко зробити доступною і для об'єктів визначених користувачами класів.

Слід також зазначити, що об'єкти можуть бути змінними (mutable) та незмінними (immutable). Наприклад, рядки в Python є незмінними, тому операції над рядками створюють нові рядки.

Карта вбудованих типів (з іменами функцій для приведення до потрібного типу і іменами класів для наслідування від цих типів):

- спеціальні типи: None, NotImplemented і Ellipsis;
- числа;
    - цілі
        - звичайне ціле int
        - ціле довільної точності long
        - логічний bool
    - число з плаваючою точкою float
    - комплексне число complex
- послідовності;
    - незмінні:
        - рядок str;
        - Unicode-рядок unicode;
        - кортеж tuple;

    - змінні:
        - список list;
- відображення:
    - словник dict
- об'єкти, які можна викликати:
    - функції (призначені для користувача і вбудовані);
    - функції-генератори;
    - методи (призначені для користувача і вбудовані);
    - класи (нові та "класичні");
    - екземпляри класів (якщо мають метод __call__);
- модулі;
- класи (див. Вище);
- екземпляри класів (див. Вище);
- файли file;
- допоміжні типи buffer, slice.

Дізнатися тип будь-якого об'єкта можна за допомогою вбудованої функції *type()*.

1.5.1. Числа

Python підтримує як цілі, так і дробові числа. Вказувати тип числа явно не потрібно, Python сам визначить його за наявністю чи відсутністю десяткової крапки.

```
>>> type(1)
<class 'int'>
```

Функцію type() можна використовувати, щоб визначати тип значення змінної. Як можна було очікувати, 1 - ціле (int).

```
>>> isinstance(1, int)
True
```

Для перевірки приналежності типу можна використовувати функцію isinstance().

```
>>> 1 + 1
2
```

Додавання двох цілих дає нам цілий результат.

```
>>> 1 + 1.0
2.0
>>> type(2.0)
<class 'float'>
```

Результатом додавання цілого та дробового числа є дробове. Для виконання додавання Python приводить ціле число до типу float і повертає результат цього ж типу.

Деякі оператори (такі, як додавання) за потреби перетворюють цілі числа в дробові. Ви можете зробити це саме самостійно.

```
>>> float(2)
2.0
```

Можна явно перетворювати int у float, використовуючи функцію float()

```
>>> int(2.0)
2
```

Дробове число можна перетворити на ціле за допомогою функції int

```
>>> int(2.5)
2
```

Функція int просто відкидає дробову частину, а не округлює.

```
>>> int(-2.5)
-2
```

Функція int для від'ємних повертає найменше ціле число, більше або рівне даному (тобто теж просто відкидає дробову частину). Тому не плутайте її з функцією math.floor.

```
>>> 1.12345678901234567890
1.1234567890123457
```

Десяткові дроби мають точність до 15 знаків після коми.

```
>>> type(1000000000000000)
<class 'int'>
```

Цілі можуть бути як завгодно великими.

Цислові операції

>>> 11 / 2
5.5

Оператор / виконує ділення чисел з плаваючою крапкою. Він повертає float, навіть якщо чисельник та знаменник цілі.

>>> 11 // 2
5

Оператор // виконує цілочисельне ділення. Коли результат додатній, ви можете вважати його відкиданням (не округленням) дробової частини звичайного ділення, але будьте обережними з цим.

>>> −11 // 2
−6

При цілочисельному діленні від'ємних чисел оператор // округлює "вгору" до найближчого цілого. Хоча, говорячи формально, він округлює вниз, тому що -6 менше за -5.

>>> 11.0 // 2
5.0

Оператор // не завжди повертає цілі. Якщо чисельник чи знаменник дробові, // повертає float, яке, щоправда, все одно округлене до найближчого цілого.

>>> 11 ** 2
121

Оператор ** означає піднесення до степеня. 11^2=121.

```
>>> 11 % 2
1
```

Оператор % повертає остачу від цілочисельного ділення.

Крім арифметичних операцій, можна використовувати операції з модуля math.

1.5.2. Тип bool

Булеві змінні приймають лише істинне чи хибне значення. Python має для цих значень дві відповідні константи: True та False, які можна використовувати для присвоєння значень змінним. Окрім констант, булеві значення можуть приймати вирази. А в деяких місцях (наприклад, в операторі if) Python навіть очікує того, що значення виразу можна буде привести до булевого типу. Такі місця називаються

булевими контекстами. Ви можете використати майже будь-який вираз в булевому контексті, і Python намагатиметься визначити його істинність. Різні типи даних мають різні правила щодо того, які значення є істинними, а які - хибними в булевому контексті.

```
for i in (False, True):
   for j in (False, True):
      print( i, j, ":", i and j, i or j, not i)
```

Результат:

```
False False : False False True
False True : False True True
True False : False True False
True True : True True False
```

1.5.3. Тип string та тип unicode

Python рядки бувають двох типів: звичайні і Unicode-рядка. Фактично рядок - це послідовність символів (в разі звичайних рядків можна сказати "послідовність байтів"). Рядки-константи можна задати в програмі за допомогою строкових літералів. Для литералів нарівні використовуються як апострофи ('), так і звичайні подвійні лапки ("). Для багаторядкових литералів можна використовувати потроєння апострофи або потроєння лапки. Керуючі послідовності всередині строкових літералів задаються зворотною косою межею (\). Приклади написання строкових літералів:

```
s1 = "строка1"
s2 = 'строка2\nз переводом строки на наступний рядок'
s3 = """строка3
переводом строки на наступний рядок """
u1 = u'\u043f\u0440\u0438\u0432\u0435\u0442'
u2 = u'Ще один приклад
```

Операціі над рядками включають конкатенацію "+", повтор "*", форматування "%". Також рядки мають велику кількість методів, деякі з яких наведено нижче. Повний набір методів (та їх необов'язкових аргументів) можна отримати в документації по Python.

```
>>> "A" + "B"
'AB'
```

```
>>> "%s %i" % ("abc", 12)
'abc 12'
>>> "A"*10
'AAAAAAAAAA'
```

1.5.4. Тип tuple

Для представлення константної послідовності (різнорідних) об'єктів використовується тип кортеж. Літерал кортежу зазвичай записується в круглих дужках, але можна, якщо не виникають неоднозначності, писати та без них. Приклади запису кортежів:

```
p = (1.2, 3.4, 0.9)
for s in "one", "two", "three":
   print (s)
one_item = (1,)
empty = ()
p1 = 1, 3, 9 # без дужок
p2 = 3, 8, 5, # кома в кінці кортежу ігнорується
```

Використовувати синтаксис кортежів можна і в лівій частині операторапривсвоювання. У цьому випадку на основі обчислених справа значень формується кортеж та зв'язується один в один з іменами в лівій частині.

```
a, b = b, a
```

1.5.5. Тип list

В Python немає масивів з довільним типом елемента. Замість них використовуються списки. Їх можна задати за допомогою літералів, що записуються в квадратних дужках, або за допомогою спискових включень.

```
lst1 = [1, 2, 3,]
lst2 = [x**2 for x in range(10) if x % 2 == 1]
lst3 = list("abcde")
```

1.5.6. Робота з послідовностями

Таблиця 1.1 – операції та методи для роботи з послідовностями

Операція	**Пояснення**
len(s)	Довжина послідовності s
x in s (x not in s)	Перевірка приналежності елемента послідовності. Повертає True або False
s + s1	Конкатенація послідовностей s та s1
s*n, n*s	Послідовність з n раз повтореної послідовності s. Якщо n <0, повертається порожня послідовність
s[i]	Повертає i -й елемент s або len (s) -i -й, якщо i <0
s[i:j:d]	Зріз з послідовності s від i до j з кроком d
min(s)	Найменший елемент s
max(s)	Найбільший елемент s
s[i] = x	i -й елемент списку s замінюється на x
s[i:j:d] = t	Зріз від i до j (з кроком d) замінюється на (список) t
del s[i:j:d]	Видалення елементів зрізу з послідовності

Таблиця 1.2 – операції та методи для роботи зі змінними послідовностями

Метод	**Пояснення**
append(x)	Додає елемент в кінець послідовності
count(x)	Рахує кількість елементів, рівних x
extend(s)	Додає в кінець даної послідовності послідовність s
index(x)	Повертає найменше i, при якому s [i] == x.
insert(i, x)	Вставляє елемент x в i -й елемент
pop(i)	Повертає i -й елемент, видаляючи його з послідовності
reverse(s)	Змінює порядок елементів s на зворотний
sort([cmpfunc])	Сортує елементи s за заданою функцією cmpfunc

1.5.7. Тип dict

Словник (хеш, асоціативний масив) - це змінна структура даних для зберігання пар ключ-значення, де значення однозначно визначається ключем. Ключем може виступати незмінний тип даних (число, рядок, кортеж і т.п.). Порядок пар ключ-значення довільний. Нижче наведено літерал для словника і приклад роботи зі словником:

```
d = {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
d0 = {0: 'zero'}
print(d[1]) # береться значення по ключу
d[0] = 0 # присвоюється значення по ключу
del d[0] # видаляється пара ключ-значення з даними ключем
print(d)
for key& val in d|items()^ # цикл по всьому словнику
   print (key, val)
for key in d.keys(): # цикл по ключам словника
   print (key, d[key])
for val in d.values(): # цикл по значенням словника
   print (val)
d.update(d0) # доповнення словника іншим словником
print (len(d)) # виводить кількість елементів ключ–значення
```

1.5.8. Тип file

Об'єкти цього типу призначені для роботи із зовнішніми даними. В простому випадку - це файл на диску. Файлові об'єкти повинні підтримувати основні методи: read(), write(), readline(), readlines(), seek(), tell(), close() і т.д.

Наступний приклад показує копіювання файлу:

```
f1 = open("file1.txt", "r")
f2 = open("file2.txt", "w")
for line in f1.readlines():
   f2.write(line)
f2.close()
f1.close()
```

Варто зауважити, що крім власне файлів в Python використовуються і файлоподібні об'єкти. В дуже багатьох функціях просто неважливо, переданий їй об'єкт типу file або іншого типу, якщо він має всі ті ж методи. Наприклад, копіювання вмісту за посиланням (URL) в файл file2.txt можна досягти, якщо замінити перший рядок на:

```
import urllib
f1 = urllib.urlopen("https://python.org")
```

1.6. Вирази.

В сучасних мовах програмування прийнято проводити більшу частину обробки даних у виразах. Синтаксис виразів у багатьох мов програмування приблизно однаковий. Пріоритет операцій показаний в наступній таблиці (в порядку зменшення). Для унарних операцій x позначає операнд. Асоціативність операцій в Python - зліва-направо, за винятком операції піднесення до степеня (**), яка читається справа наліво.

Таблиця 1.3 – Вирази мови програмування Python

lambda	лямбда-вираз
or	логічне АБО
and	логічне І
not x	логічне НІ
in, not in	перевірка приналежності
is, is not	перевірка ідентичності
<,<=,>,>=,!=,==	порівняння
|	побітовое АБО
^	побітовое виключаюче АБО
&	побітовое І
<<, >>	побітові зсуви
*, /, %	додавання і віднімання
+, -	множення, ділення, залишок
+x, -x	унарний плюс і зміна знака
~x	побітовое НЕ
**	піднесення до степеня
x.атрибут	посилання на атрибут
x [індекс]	взяття елемента за індексом
x [від: до]	виділення зрізу (від і до)
f (аргумент, ...)	визов функції
(...)	дужки або кортеж
[..]	список або спискове включення
{кл: зн ...}	словник пар ключ–значення
`вираз`	перетворення до рядка (repr)

Таким чином, порядок обчислень операндів визначається такими правилами:

- Операнд зліва обчислюється раніше операнда справа у всіх бінарних операціях, крім зведення в ступінь.
- Порівнянь виду ***a <b <c ... y <z*** фактично рівносильна: ***(a <b) and (b <c) and ... and (y <z).***
- Перед фактичним виконанням операції обчислюються потрібні для неї операнди. В більшості бінарних операцій попередньо обчислюються обидва операнда (спочатку лівий), але операції or і and, а також порівняння обчислюють таку кількість операндів, яка достатня для отримання результату.
- Аргументи функцій, вирази для списків, кортежів, словників і т.п. обчислюються зліва-направо, в порядку проходження в виразі.

Вирази завжди має результат, хоча в деяких випадках (коли вираз обчислюється заради побічних ефектів) цей результат може бути "порожнім" - None.

1.7. Імена.

Ім'я може починатися з літери (будь-якого регістра) або підкреслення, а далі допустимо використання цифр. В якості ідентифікаторів можна застосовувати ключові слова мови і небажано перевизначати вбудовані імена. Список ключових слів можна дізнатися таким чином:

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Імена, що починаються з підкреслення або двох підкреслень, мають особливий сенс. Одиночне підкреслення говорить програмісту про те, що ім'я має місцеве застосування, і не повинно використовуватися за межами модуля. Подвійним підкресленням на початку і в кінці зазвичай наділяються спеціальні імена атрибутів.

В кожній точці програми інтерпретатор "бачить" три простори імен: локальне,

глобальне і вбудоване. Простір імен - відображення з імен в об'єкти.

Для розуміння того, як Python знаходить значення деякої змінної, необхідно ввести поняття блоку коду. В Python блоком коду є те, що виконується як єдине ціле, наприклад, тіло визначення функції, класу або модуля.

Локальні імена - імена, яким присвоєно значення в даному блоці коду. Глобальні імена - імена, які визначаються на рівні блоку коду визначення модуля або ті, які явно задані в операторі global. Вбудовані імена - імена зі спеціального словника __ builtins __.

Області видимості імен можуть бути вкладеними один в одного, наприклад, всередині викликаної функції видно імена, які визначені в визиваючому коді. Змінні, які використовуються в блоці коду, але пов'язані зі значенням поза коду, називаються вільними змінними.

Так як змінну можна пов'язати з об'єктом в будь-якому місці блоку, важливо, щоб це відбулося до її використання, інакше буде з’явиться виняток NameError. Зв'язування імен зі значеннями відбувається в операторах присвоювання, from, import, в формальних аргументах функцій, при визначенні функції або класу, в другому параметрі частини except оператора try-except.

Бажано, щоб програми не залежали від таких нюансів, а для цього досить дотримуватися наступних правил:

- Завжди слід пов'язувати змінну зі значенням до її використання.
- Необхідно уникати глобальних змінних і передавати все в якості параметрів. Глобальними на рівні модуля повинні залишитися тільки і константи, імена класів і функцій.
- Ніколи не слід використовувати from модуль import * - це може привести до затінення імен з інших модулів, а всередині визначення функції просто заборонено.

Краще переробити код, ніж використовувати глобальну змінну. Звичайно, для програм, що складаються з одного модуля, це не так важливо: адже всі визначені на рівні модуля змінні глобальні.

Прибрати зв'язок імені з об'єктом можна за допомогою оператора del. В цьому випадку, якщо об'єкт не має інших посилань на нього, він буде знищений. Для управління пам'яттю в Python використовується підрахунок посилань (reference counting), для видалення наборів об'єктів з зацикленими посиланнями - збірка сміття (garbage collection).

ПРАКТИЧНА РОБОТА

Відповідно до свого номеру по списку реалізувати на мові програмування Python наступні функції, які вказані в таблиці 1.4.

Таблиця 1.4 – Варіанти для самостійної роботи

№	*f(x)*	*h*; [*a*; *b*]
1	2	3
1	$y=\ln(x)$	h=0.1; a=1; b=1.5
2	$y=1+\ln^2(x)$	h=0.1; a=0.4; b=1.0
3	$y=1+e^x$	h=0.01; a=0.5; b=0.6
4	$y=e^{x^2}/2$	h=0.2; a=2; b=3
5	$y=\cos(x)\cdot e^{-x}$	h=0.2; a=1; b=2
6	$y=1/(1+e^{-x})$	h=0.2; a=3; b=4
7	$y=\sin(x)\cdot\sinh(x)$	h=1; a=1; b=5
8	$y=0.5+\sinh^2(x)$	h=0.2; a=2; b=3
9	$y=\sqrt{x}\cdot\cosh(x)$	h=0.2; a=3; b=4
10	$y=1/(1+\cosh^2(x))$	h=0.5; a=2; b=4
11	$y=\sqrt{x}\cdot\sinh(x)$	h=1; a=1; b=5
12	$y=e^{-x}\cdot\cosh(x)$	h=1; a=1; b=4
13	$y=\ln(x^2)$	h=0.1; a=1; b=1.4
14	$y=x+\ln(x)$	h=1; a=1; b=5
15	$y=1/(1+\sin(x))$	h=π/10; a==π/6; b==π/3
16	$y=\sin(x)+\sqrt{x}$	h=π/10; a==π/6; b==π/4
17	$y=x\cdot(1-\cos(x))$	h=0.1; a=0.4; b=0.8
18	$y=e^{x+3}\,sin(x)$	h=0.5; a=0; b=2
19	$y=\cos(x)\cdot\cosh(x)$	h=1; a=1; b=5
20	$y=e^{x+1}\cdot\sinh(x)$	h=1; a=1; b=4
21	$y=10^{-2}(5+4x)-e^{x^3+4}$	h=0.1; a=-3.4; b=-1.4
22	$y=4x^3+2^{5/4}xe^{-x}$	h=1.01; a=2.4; b=10.4
23	$y=9(x^3+3.2)\cdot\mathrm{tg}(x)$	h=0.2; a=1; b=2.4
24	$y=1.2e^{x^2}+x$	h=-0.05; a=-0.75; b=-1.5
25	$y=x^{2^2}+\cos\left(2^{3/4}+x^{3/2}\right)$	h==π/3; a=14; b=19
26	$y=\left(x^{5/2}-0.8\right)\cdot\ln\left(x^2+12.7\right)$	h=0.3; a=0.25; b=5
27	$y=0.8\cdot10^{-5}(x^3+6.7)^{7/6}$	h=0.1; a=-0.5; b=0.4
28	$y=0.4+x^{2/3}\cos(x+e^x)$	h=π/10; a=5.6; b=15.4

Висновок

В даному розділі були розглянуті основи мови програмування Python 3, а саме: типи даних, основні структури даних, стандартні методи, наведені приклади створення власних функцій та роботи з файлами.

Питання для самоконтролю

1. Що із себе представляє мова програмування Python?
2. Які типи даних в мові Python ви знаєте? Наведіть приклади.
3. Які основні логічні структури присутні в Python?
4. Що таке вирази? Які вони бувають? Наведіть приклади.
5. Що таке винятки і як їх можна реалізувати в мові Python 3?
6. Яким чином можна реалізувати функцію в Python 3?

2. ОСНОВНІ СТАНДАРТНІ МОДУЛІ PYTHON.

2.1.Поняття модуля.

Якщо залишити інтерпретатор Python, а потім увійти до нього знову, усі зроблені визначення функцій та змінних буде втрачено. Тому, якщо потрібно написати дещо довшу програму, то для цього краще використовувати текстовий редактор в якому виконану програму потрібно записати у файл. Це називається створенням скрипта. Коли програма збільшується, для полегшення її підтримки слід розбити її на кілька файлів.

Для підтримки цього, Python має певний механізм для створення визначень у файлі, які згодом можуть використовуватися у скрипті чи у діалоговому режимі інтерпретатора. Такий файл зветься *модулем*. Визначення, задані в модулі, імпортуються в інші модулі або в основний (*main*) модуль, який являє собою сукупність об'єктів.

Модуль – це файл, що складається з описів функцій та інструкцій Python. Назва файла є назвою модуля, до якої додається розширення .py. Всередині модуля його назва доступна через значення глобальної змінної __*name*__. Для прикладу, створимо у текстовому редакторі файл *fibo.py* у поточній директорії з таким змістом:

```
# Модуль, що обчислює числа Фібоначчі
def fib(n): # виводить числа Фібоначчі до n
   a, b = 0, 1
   while b < n:
      print(b)
      a, b = b, a+b
def fib2(n): # повертає числа Фібоначчі до n
   result = []
   a, b = 0, 1
   while b < n:
      result.append(b)
      a, b = b, a+b
   return result
```

Тепер відкриємо інтерпретатор Python та імпортуємо цей модуль за допомогою такої команди:

>>> import fibo

Ця команда додає в поточний простір імен лише саму назву модуля *fibo*, а не назви функцій, визначених у ньому. Використовуючи назву модуля ви маєте доступ до його функцій:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Якщо часто використовувати функцію, то їй краще дати локальну назву:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

2.2.Модулі в python.

Модуль може містити не лише описи функцій, а й виконувані інструкції. Ці інструкції потрібні для ініціалізації модуля. Вони виконуються лише при першому імпортуванні модуля. (Насправді визначення функцій – це також інструкції, що "виконуються"; виконання ж полягає в тому, що назва функції вводиться у глобальний простір імен модуля).

Кожен модуль має свій власний простір імен, який використовується як глобальний усіма визначеними у цьому модулі функціями. Таким чином, автор модуля може використовувати глобальні змінні всередині модуля, уникнувши при цьому можливого конфлікту з глобальними змінними, що задані користувачем модуля. З іншого боку, ви можете дістатися до глобальних змінних модуля за допомогою тієї ж нотації, що використовується для доступу функцій:

назва_модуля.елемент_модуля

Модулі можуть імпортувати інші модулі. Зазвичай, хоча це і не є необхідним, всі інструкції *import* пишуть на початку модуля. Імпортовані назви модулів

додаються до глобального простору імен модуля.

Існує варіант інструкції *import*, який напряму імпортує назви з модуля у простір імен імпортуючого модуля. Наприклад:

>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

Ця операція не додає до локальної символьної таблиці назву самого модуля, з якого відбувся імпорт (зокрема, у цьому прикладі назва fibo – невизначена).

Існує також можливість для імпорту всіх назв, визначених у модулі:

>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

При цьому імпортуються усі назви, крім тих, що починаються з символу підкреслювання (_).

2.2.1. Шлях пошуку модулів

Якщо імпортується модуль, який називається *spam*, то інтерпретатор спочатку шукає файл з назвою *spam.py* у поточній директорії, а потім у директоріях, визначених змінною середовища *PYTHONPATH*. Вона має такий же синтаксис, як і змінна оболонки *PATH*, тобто являє собою список директорій. Якщо *PYTHONPATH* не задано, або якщо файл там не знайдено, то пошук продувжується за типовою адресою, яка залежить від інсталяції; у системах *Unix* це здебільшого */usr/local/lib/python*.

Тут слід уточнити, що пошук модулів починається зі списку директорій, заданих змінною *sys.path*, яка ініціалізується директорією, де розміщено скрипт вводу (чи у поточній директорії), а потім доповнюється з *PYTHONPATH* та залежним від інсталяції шляхом. Це дозволяє програмам змінювати адресу пошуку. Зауважимо, що оскільки назва директорії, де знаходиться виконуваний скрипт, є у змінній *sys.path*, тому важливо, щоб назва скрипта не збігалася з назвою певного стандартного модуля, бо інакше Python спробує завантажити скрипт замість модуля при імпорті. Загалом це повинно призвести до помилки.

2.2.2. Компільовані файли

Існує можливість значного прискорення запуску коротких програм, що використовують багато стандартних модулів: якщо в директорії, де знаходиться файл *spam.py*, існує файл *spam.pyc*, то вважається, що він містить скомпільовану в байт-код версію модуля *spam*. Час останьої зміни версії *spam.py*, що використовується для створення *spam.pyc*, записується у *spam.pyc*, і файл *.pyc* пропускається, якщо час зміни скомпільованої версії не відповідає текстовій.

У більшості випадків для створення файла *spam.pyc* взагалі не потрібно нічого робити. Як тільки *spam.py* скомпільовано, інтерпретатор зробить спробу записати скомпільовану версію у *spam.pyc*. Якщо ця спроба не вдається, то це не призводить до помилки. Якщо з певних причин файл не записано повністю, то новостворений *spam.pyc* буде вважатися недійсним і таким чином не буде використовуватися пізніше. Вміст файла *spam.pyc* не залежить від платформи, отже директорія, що містить модулі, написані на Python, може використовуватися машинами різних архітектур.

2.2.3. Стандартні модулі

Python має бібліотеку стандартних модулів, яка описана в окремому документі, що зветься "Довідник бібліотеки мови Python" (Python Library Reference) (надалі "Довідник бібліотеки"). Окремі модулі, що вбудовано в інтерпретатор, надають доступ до операцій, що не є основною частиною мови, але все таки вони вбудовані, або для ефективності, або для того, щоб надати доступ до функцій операційної системи, зокрема системних викликів. Такі модулі є частиною конфігурації, яка залежить від платформи.

Один окремий модуль заслуговує спеціальної уваги: *sys*, який вбудовано у будь-який інтерпретатор мови Python.

Змінна *sys.path* – це список рядків, що визначають використовувані інтерпретатором шляхи пошуку модулів. Вона ініціалізується стандартним значенням, якее можна дістати із змінної середовища *PYTHONPATH* або із вбудованого стандартного значення (якщо *PYTHONPATH* не задано). Її можна змінити за допомогою стандартних операцій зі списками:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

Вбудована функція *dir()* використовується для виявлення усіх назв, визначених у модулі. Вона повертає впорядкований список рядків:

```
>>> import fibo, sys
>>> dir(fibo)
```

Без аргументів *dir()* повертає назви, визначені на момент виклику функції:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Зауважимо, що цей список містить всі види ідентифікаторів: змінні, модулі, функції тощо.

dir() не видає назв вбудованих функцій та змінних. Якщо ці назви потрібні, то вони визначені у стандартному модулі *__builtin__*:

```
>>> import __builtin__
>>> dir(__builtin__)
```

2.3. Огляд стандартної бібліотеки.

2.3.1. Інтерфейс операційної системи

Модуль *os* містить функції взаємодії з операційною системою:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd() # Повертає поточну робочу директорію
'C:\\Python24'
```

```
>>> os.chdir('/server/accesslogs')
```

Пам'ятайте, що слід використовувати "*import os*" замість "*from os import *"*. Це дозволить запобігти перекриванню вбудовано функції *open()* функцією *os.open(),* яка має зовсім інше призначення.

Вбудовані функції *dir()* та *help()* є дуже корисними для отримання допомоги при роботі з такими великими модулями як *os*:

```
>>> import os
>>> dir(os) # повертає список усіх функцій модуля
>>> help(os) # повертає інструкцію створену збиранням до купи рядків документації модуля
```

Для щоденних потреб, пов'язаних з файлами та директоріями, модуль *shutil* надає інтерфейс більш високого рівня, що спрощує програмування:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db') # копіювання
>>> shutil.move('/build/executables', 'installdir') # переміщення
```

2.3.2. Шаблони розширення файлових назв

Модуль *glob* містить функцію, що дозволяє створювати списки файлів за допомогою шаблонів розширення, застосованих до директорій:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

2.3.3. Аргументи командного рядка

Скрипти часто використовують аргументи, подані з командного рядка. Ці аргументи зберігаються у вигляді списку атрибута *argv*, що знаходиться в модулі *sys*. Наприклад, якщо з командного рядка було запущено команду "*python demo.py one two three*", то ми можемо отримати такий вивід:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Модуль *getopt* оброблює *sys.argv* на основі конвенцій юніксової функції *getopt()*. Потужнішу і гнучкішу обробку командного рядка можна знайти у модулі *optparse*.

2.3.4. Переспрямування виводу помилок та вихід із програми

Модуль *sys* має також атрибути *stdin*, *stdout* та *stderr* ("стандартний ввід", "стандартний вивід" та "стандартний вивід помилок" відповідно). Останній корисний для виводу попереджень і помилок при переспрямуванні *stdout*:

```
>>> sys.stderr.write('Попередження: файл для запису не знайдено; створюється новий файл')
Попередження: файл для запису не знайдено; створюється новий файл
```

Найпростіший шлях виходу з програми — це виклик "*sys.exit()*".

2.3.5. Пошук за шаблоном

Модуль *re* містить утиліти регулярних виразів для пошуку за шаблоном всередині рядків. Регулярні вирази надають компактні оптимальні вирішення при застосуванні доволі складних правил пошуку:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Якщо потрібні лише прості маніпуляції, то найкраще застосовувати методи рядків, які набагато простіше читати:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

2.3.6. Математика

Модуль *math* надає можливість доступу до функцій бібліотеки С для роботи з дробовими числами:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Модуль *random* містить утиліти для роботи з випадковими числами:

```
>>> import random
>>> print random.choice(['яблуко', 'груша', 'банан'])
```

```
'яблуко'
>>> random.sample(xrange(100), 10)   # вибір без заміщення
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # випадкове число з рухомою комою
0.17970987693706186
>>> random.randrange(6)    # випадкове ціле число, вибране з послідовності range(6)
4
```

2.3.7. Доступ до мережі Інтернет

Існують кілька модулів для доступу до інтернету та обробки його протоколів. Два найпростіші — це *urllib* (для отримання даних з інтернет-адрес) та *smtplib* для відправлення електронної пошти:

```
>>> import urllib.request as req
>>> for line in req.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... if 'EST' in line:      # шукаємо Eastern Standard Time
...     print line

Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org', 'jceasar@tmp.org',
"""To: jceasar@tmp.org
From: soothsayer@tmp.org

Beware the Ides of March.
""")
>>> server.quit()
```

2.3.8. Час і число

Модуль *datetime* містить класи для роботи з даними, що виражають час та число, як у складний так і в простий спосіб. Він придатний і для арифметики часових даних, хоча основна увага приділяється тому, щоб ефективно дістати дані для форматування та їхньої обробки. Модуль також має об'єкти, що розрізняють різні часові зони.

```
# створення та форматування чисел дуже просте
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December'

# часові дані придатні для застосування календарної арифметики
>>> birthday = date(1964, 7, 31)
>>> age = now — birthday
>>> age.days
14368
```

2.3.9. Ущільнення даних

Поширені формати ущільнення та архівації даних напряму підтримуються такими модулями як *zlib*, *gzip*, *bz2*, *zipfile* та *tarfile*.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031
```

2.3.10. Обчислення продуктивності

Окремі користувачі мови Python зацікавлені у тому, наскільки продуктивними є різні підходи вирішення однієї проблеми відносно один одного. До складу Python входить модуль *timeit*, який дозволяє швидко віднайти відповіді на ці питання.

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.60864915603680925
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
```

0.8625194857439773

На відміну від високого рівня детальності модуля *timeit*, модулі *profile* та *pstats* мають інструменти для ідентифікації критичних ділянок коду у більших блоках коду.

2.3.11. Контроль якості

Один із способів для створення якісного програмного забезпечення - це написання спеціальних тестів для кожної функції, і часте використання цих тестів під час процесу розробки.

Модуль *doctest* має спеціальні інструменти для сканування модуля та перевірки тестів, що вказані в рядках документації. Створення ж тестів – дуже просте і полягає у копіюванні та вставці типового виклику функції та її результату в рядок документації. Додання прикладу вдосконалює документацію а також дозволяє модулю *doctest* перевірити, чи відповідає код документації:

```
def average(values):
    """Виводить середнє арифметичне для даного списку чисел.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()   # автоматично перевірити тести
```

Модуль *unittest* є дещо складнішим за *doctest*, але натомість дозволяє провести більш ґрунтовне тестування за допомогою правил, що здебільшого задаються в окремому файлі:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
```

```
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Виклик з командного рядка запускає всі тести
```

2.3.12. Форматування виводу

Модуль *repr* має версію функції *repr()* для скороченого зображення великих або багаторівневих структур даних:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Модуль *pprint* (pretty printer) надає можливість більш досконалого контролю при виводі об'єктів (як вбудованих, так і заданих користувачем) у вигляді, придатному для зчитування інтерпретатором. Якщо результат довший за один рядок, то ця функція додає пробіли та символи нового рядка, які дозволяють ясніше виразити структуру даних:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

Модуль *textwrap* форматує текст для певної ширини екрану:

```
>>> import textwrap
>>> doc = """Метод wrap() подібний до fill(), але він повертає
... список рядків замість одного довгого рядка, розбитого
... на рядки."""
...
>>> print textwrap.fill(doc, width=40)
Метод wrap() подібний до fill(),
```

але він повертає список рядків
замість одного довгого рядка,
розбитого на рядки.

Модуль *locale* завантажує формати даних, специфічні для певного культурного оточення. Спеціальний атрибут форматуючої функції модуля надає можливість прямого форматування чисел за допомогою групових роздільників:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'uk_UA.utf8')
('uk_UA', 'utf8')
>>> conv = locale.localeconv()   # отримати правила переведення
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
>>>  print locale.format("%.*f%s",
...        (conv['int_frac_digits'], x,
...        conv['currency_symbol']), grouping=True)
1.234.567,80гр
```

2.3.13. Шаблони

Модуль *string* має клас *Template* з досить простим синтаксисом, придатним для редагування користувачами. За його допомогою користувачі можуть змінювати текстові величини програми без внесення змін до її коду.

Формат модуля використовує спеціальні назви-заповнювачі, що утворюються за допомогою символа "*$*" та дійсного ідентифікатора мови Python (буквено-цифрові символи та нижня риска). Для відокремлення назви-заповнювача від наступних буквено-цифрових символів слід використовувати фігурні дужки. "*$$*" задає один символ "*$*":

```
>>> from string import Template
>>> t = Template('${village} витратили $$10 на $cause.')
>>> print t.substitute(village='Васюки', cause='сміттєфонд')
Васюки витратили $10 на сміттєфонд.
```

Метод *substitute* відкидає *KeyError,* якщо ключове слово не існує в словнику або в ключовому аргументі. Для програм, де дані для заповнення шаблонів можуть бути неповними, краще використовувати метод *safe_substitute*, що за умови

відсутності відповідних даних залишить незаповнені назви без змін.

```
>>> t = Template('Повернути $item $owner.')
>>> d = dict(item='непроковтнутий шматок')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> print t.safe_substitute(d)
Повернути непроковтнутий шматок $owner.
```

Детальніше ознайомитися з усіма стандартними бібліотеками Python можна за посиланням https://docs.python.org/3/library/index.html .

ПРАКТИЧНА РОБОТА

1) Відкриття файлу для зчитування

```
def read_file(fname):
"""Функція для зчитування файлу fname
та виведення його вмісту на екран"""
   file=open(fname, 'r') # відкриття файлу для зчитування
   print('File '+fname+':') # виведення назви файлу
   # Зчитування вмісту файлу по рядкам
   for line in file:
   # Виведення рядка s
      print(line, end='')
file.close() # Закриття файлу
if __name__=='__main__': read_file('data/file.txt')
```

2) Використання функції *os.path.join* для побудови шляху до файлу

```
# Модуль, який містить функції для роботи з шляхом у файловій
системі
import os.path

def read_file(fname):
   """Функція для зчитування файла fname
   та виведення його вмісту на екран"""
   file=open(fname,'r') # відкриття файлу для зчитування
   print('File '+fname+':') # виведення назви файлу
```

```
    # зчитування вмісту файла по рядках
    for line in file:
        print(line, end='') # виведення рядка s
file.close() # закриття файлу
if __name__=='__main__':
# функція os.path.join з'єднує частини шляху у файловій системі
# необхідним роздільником
    read_file(os.path.join('data', 'file.txt'))
```

3) Запис даних у текстовий файл

```
import os.path
text='''Hello!
I am a text file. And I had been written with a Python script
before you opened me, so look up the docs and try to delete
me using Python, too.'''
def write_text_to_file(filename, text):
    """Функція для запису у файл filename рядка text"""
    f=open(filename, "w")# відкриття файла для запису
    f.write(text) # Запис рядка text у файл
    f.close()# Закриття файлу
if __name__=='__main__':
    write_text_to_file(os.path.join('data','example02.txt'),text)
```

4) Використання оператора *with* для закриття файлу

```
import os.path
filename=os.path.join('data','file.txt') #побудова імені файлу
# Оператор with закриє файл по закінченню виконання
# операторів усередині нього або виникненні виключення
with open(filename) as file: print(file.read())
```

5) Перезапис файла

```
import os.path
filename=os.path.join('data', 'example07.txt')
# Зчитування файла
with open(filename, 'r') as file:
    lines=file.readlines()
# Модифікація даних
lines.insert(2, 'inserted line\n')
# Перезапис файла
with open(filename, 'w') as file:
    file.writelines(lines)
```

Висновок

В даному розділі були розглянуті основи роботи з модулями та стандартні модулі мови програмування Python 3. Пройшовши даний розділ студент зможе:

1) визначати поточний шлях до заданого модуля;
2) реалізовувати регулярні вирази;
3) знати стандартні методи роботи з часом, мережею;
4) підключати бібліотеку для простих математичних обчислень;
5) написати тести.

Питання для самоконтролю

1. Що таке модуль?
2. Для чого використовують PATH/PYTHONPATH?
3. Що таке os/sys?
4. Яким чином реалізувати регулярний вираз в мові Python 3? Наведіть приклад.
5. За допомогою чого в Python 3 можна: виконувати математичні обчислення, працювати з датою, часом, мережею, архівованими даними?
6. Яким чином можна реалізувати тести в Python 3?

3. ЕЛЕМЕНТИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ.Що таке функціональне програмування?

Основна перевага використання функцій – це можливість повторного застосування програмного коду, тобто, їх можна викликати багато разів не тільки в тій програмі, де її було визначено, але, можливо, і в інших програмах, іншими користувачами та для інших цілей.

Функціональне програмування (далі ФП) - це стиль програмування,використовує тільки композиції функцій. Іншими словами, це програмування за допомогою виразів, а не імперативних команд.

Функціональне програмування має такі властивості:

- Функції – об'єкти першого класу. Тобто, усе, що можна робити з "даними", можна робити і з функціями.
- Рекурсія як основна структура керування.
- Акцент на обробці списків (Lisp – LISt Processing).
- Функціональні мови унеможливлюють побічні ефекти.
- ФП не схвалює застосування операторів (statements). Замість них – обчислення виразів (тобто функцій з аргументами).
- ФП наголошує на тому, що має бути обчислено, а не як.
- У ФП переважно використовують функції вищого порядку – функції, що оперують функціями

3.2. Функціональна програма.

В математиці функція відображає об'єкти з однієї множини (множини визначення функції) в іншу (множина значень функції). Математичні функції (їх називають чистими) "механічно", однозначно обчислюють результат по заданим аргументам. Чисті функції не повинні зберігати в собі будь-які дані між двома викликами. Чисту функцію можна представити у вигляді чорного ящика, про яку відомо тільки те, що вона робить, але зовсім не важливо, як.

Програми в функціональному стилі конструюються як композиція функцій. При цьому функції розуміються майже так само, як і в математиці: вони відображають одні об'єкти в інші. У програмуванні "чисті" функції - ідеал, не

завжди досяжний на практиці. Практично корисні функції зазвичай мають побічний ефект: зберігають стан між викликами або змінюють стан інших об'єктів. Наприклад, без побічних ефектів неможливо уявити собі функції введення-виведення. Власне, такі функції заради цих "ефектів" і використовуються. Крім того, математичні функції легко працюють з об'єктами, які вимагають нескінченного обсягу інформації (наприклад, дійсні числа). У загальному випадку комп'ютерна програма може виконати лише наближені обчислення.

3.3. Функція: визначення і виклик.

Створімо функцію, що виводить числа Фібоначчі до певної межі:

```
def fib(n): # вивести числа Фібоначчі до n
    """Вивести числа Фібоначчі до n."""
    a, b = 0, 1
    while b < n:
        print (b)
        a, b = b, a+b
# Тепер викликаємо щойно задану функцію:
fib(2000)
```

Ключове слово *def* вводить визначення (definition) функції. За ним повинна бути назва функції та оточений дужками список формальних параметрів. Інструкції які утворюють тіло функції починаються з наступного рядка і повинні бути виділені пробілами. Першим, але необов'язковим, рядком функції може бути символьна константа, яка коротко описує функцію. Її називають рядком документації.

Існують спеціальні утиліти, що використовують документаційні рядки для автоматичного створення друкованої чи онлайн документації або для перегляду коду в діалоговому режимі.

Виконання функції вводить новий простір імен, що використовується для локальних змінних функції. Зокрема, всі присвоєння змінним всередині функції зберігають свої значення у локальному просторі імен, тоді як при посиланні на змінну пошук починається у локальному, а потім продовжується у глобальному, і

наприкінці – у просторі імен вбудованих ідентифікаторів. Таким чином, глобальні змінні не можуть отримувати нові значення всередині функцій (за винятком якщо вони названі у твердженні global), хоча посилання на них можливе.

Параметри (або аргументи) функції вводяться в простір імен функції при її виклику. Аргументи передаються за значенням, де значення - завжди посилання на об'єкт, а не значення самого об'єкта, тому точніше було б сказати - передача за посиланням. При передачі змінюваного об'єкта будь-які його зміни вседедині викликаної функції стануть видимі в середовищі, що викликало цю функцію, наприклад, додавання нових елементів до списку. Коли одна функція викликає іншу – створюється новий локальний простір імен для цього виклику.

Визначення функції додає назву функції до поточного простору імен. Значення назви функції належить до типу, який ідентифікується інтерпретатором як задана користувачем функція. Це значення може присвоюватися іншій змінній, що потім теж може використовуватися як функція. Тут показано, як діє загальний механізм перейменування:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

У Python, як і в C, процедури – це функції, що не повертають жодного значення. Власне, з технічної точки зору, процедури таки повертають певне значення None. Зазвичай, це значення не виводиться інтерпретатором, якщо це єдине можливе значення для виводу.

```
>>> print (fib(0))
None
```

Створення функції, що повертає список чисел Фібоначчі замість виведення їх на друк, досить просте:

```
def fib2(n): # повертає числа Фібоначчі до n
    """Повертає список чисел Фібоначчі до n"""
    result = []
```

```
    a, b = 0, 1
    while b < n:
        result.append(b) # див. нижче
        a, b = b, a+b
    return result

f100 = fib2(100) # виклик функції
>>>f100 # вивід результату
 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Цей приклад також демонструє кілька нових властивостей мови Python:

- Оператор *return* повертає із функції певне значення. Якщо *return* вжито без аргументів, то результатом повернення є *None*. Якщо процедура закінчується сама по собі, то результатом повернення є *None*.

- Інструкція *result.append(b)* викликає метод об'єкта списку *result*. *Метод* - це функція, що "належить" певному об'єкту. Вона викликається у формі *obj.назва_методу*, де *obj* — певний об'єкт (може також бути виразом) і назва_методу — назва методу, визначеного типом об'єкта. Різні типи визначають різні методи. Методи різних типів можуть мати однакову назву, не спричиняючи при цьому двозначності. Наведений у цьому прикладі метод *append()* визначений для спискових об'єктів; він додає новий елемент в кінець списку. У цьому прикладі це еквівалентно "result = result + [b]", але цей метод є більш ефективним.

Можливо також визначити функцію зі змінною кількістю аргументів. Для цього існує три способи, що можуть сполучуватися.

3.3.1. Стандартні значення аргументів

Найкорисніший спосіб — це визначити типове значення для одного чи кількох аргументів. Це створює можливість виклику функції з меншою кількістю аргументів, аніж задано у визначенні функції. Наприклад:

```
def ask_ok(prompt, retries=4, complaint='Так чи ні, будь-ласка!'):
    while True:
        ok = input(prompt)
        if ok in ('т', 'та', 'так'): return True
```

```
        if ok in ('н', 'ні', 'нєт'): return False
        retries = retries - 1
        if retries < 0: raise IOError('затятий користувач')
        print (complaint)
```

Ця функція може викликатися як *ask_ok('Справді закрити програму?')* чи як *ask_ok('Переписати файли?', 2)*.

Цей приклад також ілюструє ключове слово *in*, що дозволяє перевірити чи дана послідовність містить певний елемент.

Стандартні значення обчислюються в момент задання функції відповідно до визначаючого контенсту, тому:

```
i = 5
def f(arg=i):
   print arg
i = 6
f()
```

Виведе 5.

Важливе зауваження: стандартнне значення обчислюється лише раз. Хоча існує виняток, коли стандартнне значення — змінюваний об'єкт, скажімо, список, словник, реалізація більшості класів. Наприклад, наступна функція акумулює аргументи, що передаються при подальших викликах:

```
def f(a, L=[]):
   L.append(a)
   return L

print (f(1))
print (f(2))
print (f(3))
```

Виведе:

```
[1]
[1, 2]
[1, 2, 3]
```

Якщо потрібно, щоб стандартне значення було спільним для всіх наступних викликів, можна створити функцію на зразок:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print (f(1))
print (f(2))
print (f(3))
```

3.3.2. Ключові аргументи

Функції можуть також викликатися за допомогою ключових аргументів у вигляді "*ключ = значення*". Наприклад:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print ("-- This parrot wouldn't", action)
    print ("if you put", voltage, "Volts through it.")
    print ("-- Lovely plumage, the", type)
    print ("-- It's", state, "!")
```

може викликатися у будь-який вказаний нижче спосіб:

```
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

але такі виклики неправильні:

```
parrot() # пропущено обов'язковий аргумент
parrot(voltage=5.0, 'dead') # неключовий аргумент передається після ключового
parrot(110, voltage=220) # два значення для одного аргумента
parrot(actor='John Cleese') # невідомий ключ
```

Взагалі в списку аргументів позиційні аргументи повинні бути розташовані перед ключовими, при цьому ключі повинні бути вибрані з формальних назв параметрів. Чи задані для цього параметра стандартні значення — не важливо. Жоден аргумент не може отримувати значення більш, ніж один раз — формальні назви параметрів, що відповідають позиційним аргументам не можуть використовуватися як ключові слова під час того самого виклику. Ось приклад того, коли помилка відбувається саме через це обмеження:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Якщо останній формальний параметр задано у формі **назва, то він отримує словник, що складається з аргументів, чиї ключі відповідають формальним параметрам. Він може сполучатися з формальним параметром у формі *назва. який отримує кортеж, що складається з позиційних аргументів, не включених у список формальних пареметрів (аргумент *назва повинен передувати аргументу **назва). Наприклад, функцію, задано таким чином:

```
def cheeseshop(kind, *arguments, **keywords):
   print ("-- Do you have any", kind, '?')
   print ("-- I'm sorry, we're all out of", kind)
   for arg in arguments: print (arg)
   print ('-'*40)
   keys = keywords.keys()
   for kw in keys: print (kw, ':', keywords[kw])
```

Виведе:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
```

It's very runny, sir.
It's really very, VERY runny, sir.
--
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

3.3.3. Списки аргументів довільної довжини

Аргументи передаються за допомогою кортежа. Нуль чи більше звичайних аргументів можуть передувати змінній кількості аргументів.

```
def fprintf(file, format, *args):
   file.write(format % args)
```

Розпакування списків аргументів

Зворотня ситуація трапляється, коли аргументи задані списком чи кортежем, але їх потрібно розпакувати для виклику функції, що потребує окремих позиційних аргументів. Наприклад, вбудована функція *range()* потребує двох окремих аргументів, що вказують на межі послідовності. Якщо вони не задані окремо, виклик функції слід писати з оператором *, що дозволяє розпакувати аргументи, задані списком чи кортежем:

```
>>> range(3, 6) # звичайний виклик з окремими аргументами
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args) # виклик із аргументами, розпакованими зі списку
[3, 4, 5]
```

Лямбда-функції

За популярною вимогою до Python було додано кілька нових властивостей, типових для функціональних мов програмування та мови Lisp. Ключове слово *lambda* дозволяє створювати невеличкі анонімні функції. Ось, наприклад, функція, що повертає суму двох своїх аргументів: "*lambda a, b: a+b*". Лямбда-функції можуть стати в нагоді, коли потрібні об'єкти функцій. Подібно до вкладених функцій

лямбда-функції можуть посилатися на змінні із зовнішнього контексту:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

3.4. Рекурсія.

Кожна з функцій може викликати інші функції, у тому числі звертатися до самої себе. Функцію, яка звертається до самої себе, називають рекурсивною. *Рекурсія* – виклик функції з неї ж самої, безпосередньо (пряма) або через інші функції (складна або непряма): коли функція first викликає функцію second, а та у свою чергу викликає функцію first. Розглянемо рекурсію на прикладі перетворення десяткового числа у список з розрядів його двійкового відображення:

```
def bin(n):
    digits = []
    while n > 0:
        n, d = divmod(n, 2)
        digits = [d] + digits
    return digits

print (bin(69))
```

Тепер перепишемо функцію ***bin*** використовуючи рекурсію:

```
def bin(n):
    if n == 0:
        return []
    n, d = divmod(n, 2)
    return bin(n) + [d]

print (bin(69))
```

Вище показано, що цикл while більше не використовується, а замість нього з'явилося умова закінчення рекурсії: умова, при виконанні якого функція не викликає себе.

Кількість вкладених викликів функції називають глибиною рекурсії.

Обов‘язковим елементом в описі будь-якого рекурсивного процесу є деяке твердження (оператор), який визначає умову завершення рекурсії (іноді його називають опорною умовою). Тут можна задати певне фіксоване значення, яке обов‘язково буде досягнуто під час рекурсивного обчислення, дозволяючи організувати своєчасне призупинення процесу. Крім того, повинен існувати спосіб зображення одного кроку розв‘язання за допомогою іншого, простішого. Кількість рівнів вкладеності не обмежена і може бути досить великою. Кожен виклик рекурсивної функції повинен наближати випадок, який зупиняє рекурсивні виклики, інакше виконання функції ніколи не припиниться через нескінченний ланцюжок рекурсивних викликів. Найпростіший спосіб гарантувати виконання опорної умови є зменшення деякої додатної величини
до досягнення деякого «малого» значення.

Особливістю реалізації рекурсивних функцій є те, що у програмі існує тільки один екземпляр коду цієї функції. На кожному рівні рекурсії здійснюється нове звертання до цього коду. Рис. 3.1 ілюструє виконання рекурсивного процесу з трьома рівнями рекурсивного виклику, правда для простоти пояснення вважається, що на кожному рівні створюється новий екземпляр коду. Розглянемо послідовність дій, які виконують у цьому випадку.

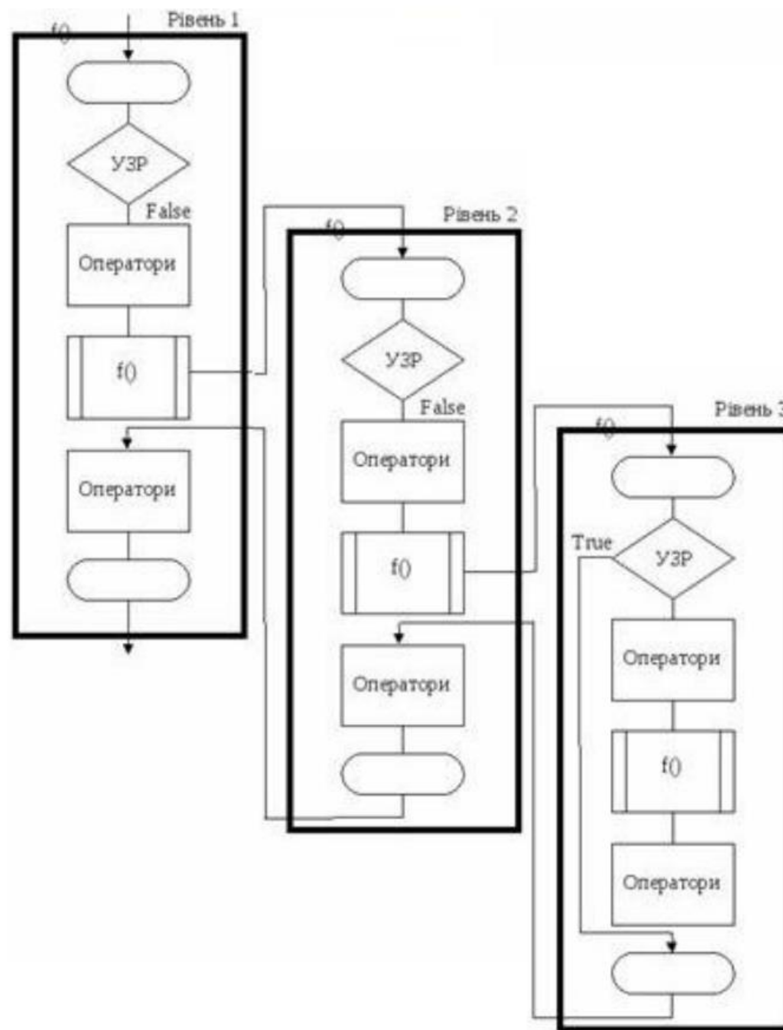

Рис. 3.1 – Схема виконання рекурсивного процесу (УЗР– умова завершення рекурсії)

Виклик функції забезпечує початок виконання її коду. Припускаючи, що на першому рівні рекурсії опорна умова не виконується, маємо виконання операторів, розташованих за опорною умовою, і новий виклик функції.

На цьому виконання функції на першому рівні тимчасово переривається, і розпочинається другий рівень рекурсії. Другий рівень рекурсії (рис.3.1) є аналогом першому. Він також тимчасово переривається з переходом до третього рівня рекурсії. Якщо припустити, що на третьому рівні рекурсії опорна умова виконана, маємо завершення третього рівня з поверненням на другий рівень у точку, де відбулося тимчасове переривання виконання процесу. Далі завершуються обчислення на другому рівні рекурсії з наступним поверненням на перший рівень. Після завершення обчислень на першому рівні, отримуємо остаточний результат.

Рекурсивні функції можуть вимагати великих обсягів пам‘яті для свого виконання. Це пояснюється тим, що, як і у випадку звичайних функцій, локальні змінні створюються в програмному стеку на кожному рівні рекурсивного виклику, що може привести до ситуації, коли обсяг стека вичерпається, оскільки знищення локальних змінних здійснюється тільки при виході з функції.

Будь–яку рекурсивну функцію можна замінити циклом.

Функції, які виконують один рекурсивний виклик на кожній рекурсивній гілці. Структурно рекурсивна функція на верхньому рівні завжди має команду розгалуження у вигляді вибору однієї з двох або більше альтернатив в залежності від умови (умов), яку в цьому випадку доречно назвати «умовою припинення рекурсії». Умова має дві або більше альтернативні гілки, з яких хоча б одна є рекурсивною і хоча б одна – термінальною.

Рекурсивну гілку виконують, коли умова припинення рекурсії має значення «хибність», і містить хоча б один рекурсивний виклик – прямий або опосередкований виклик функцією самої себе.

Термінальну гілку виконують, коли умова припинення рекурсії має значення «істина» (вона повертає деяке значення, не виконуючи рекурсивного виклику). Правильно написана рекурсивна функція повинна гарантувати, що через скінчену кількість рекурсивних викликів буде досягнуто виконання умови припинення рекурсії, в результаті чого ланцюжок послідовних рекурсивних викликів буде перерваний.

Бувають випадки «паралельної рекурсії», коли на одній рекурсивній гілці виконують два або більше рекурсивних виклики. Паралельна рекурсія типова при обробці складних структур даних, таких як дерева. Найпростіший приклад паралельно-рекурсивної функції – обчислення ряду Фібоначчі, де для отримання значення n-го члена необхідно обчислити (n–1)-й і (n–2)-й.

```
def Fib(n):
   if n < 2:
      return n
   else:
```

```
        return Fib(n-1) + Fib(n-2)
    print(Fib(33))
```

Питання про бажаність використання рекурсивних функцій в програмуванні є неоднозначним. З одного боку, рекурсивна форма може бути структурно простішою і наочнішою, особливо, коли сам алгоритм, по суті є рекурсивним. З іншого боку, рекомендують уникати рекурсивних програм, які призводять (або в деяких умовах можуть призводити) до рекурсії великої глибини. Приклад рекурсивного обчислення факторіала є, скоріше, прикладом того, як не треба застосовувати рекурсію, тому що призводить до досить великої глибині рекурсії і має очевидну реалізацію у вигляді звичайного циклічного алгоритму.

ПРАКТИЧНА РОБОТА

1) Визначеня функції hello_world

```
def hello_world():
   print('Hello, World!')
# Виклик функції
     hello_world()
```

2) Формальний параметр функції

```
def print_numbers(limit):
   for i in range(limit): print(i)
# Виклик функції print_numbers, її формальний
# параметр limit замінюють фактичним параметром 10
print_numbers(10)
```

3)
```
def print_numbers(limit):
    for i in range(limit): print(i)
# Виклик функції з фактичним параметром n
```

```
# print_numbers
n = int(input('Введіть n: '))
print_numbers(n)
```

4)
```
def print_numbers(limit):
    for i in range(limit): print(i)
def main():
    n=int(input('Введіть n: ')); print_numbers(n)
# Виклик функції
main()
```

5)
```
def add_numbers(a, b):
    return a + b # вихід функції - сума параметрів
# Виклик функції
x = add_numbers(2, 3);print(x)
```

6) Функції можна передавати будь-яку кількість позиційних та іменованих аргументів. Для цього перед ім'ям заданого словника в списку формальних параметрів ставиться два символи **

```
def function(**kwargs):
    print(kwargs)
function (arg1='value1', arg2='value2')
# Можна розпакувати відображення
# в іменовані параметри при виклику функції
options = {
    'sep': ', ',
    'end': ';\n'}
print('value1', 'value2', **options)
```

Результат:

{'arg1': 'value1', 'arg2': 'value2'}

value1, value2;

7) Якщо параметр не задано

```
def hello(name='Alex'):
   print('Hello, ', name, '!', sep='')
# Виклик функції
hello('Python'); hello()
```

Висновок

В даному розділі були розглянуті основи фунціонального програмування в Python 3. Пройшовши даний розділ студент зможе:

1) визначати та викликати функцію;
2) визначати функцію з дефолтними значеннями аргументів та різною їх кількістю;
3) визначати та вміти користуватись при необхідності лямбда-функціями;
4) розуміти та використовувати рекурсивний підхід в програмуванні.

Питання для самоконтролю

1. Що таке функціональне програмування?
2. Як визначити викликати функцію?
3. Як задати дефолтне значення для одного/декількох аргументів функції?
4. Що таке ключові аргументи?
5. Як створити функцію з довільним числом аргументів?
6. Що таке лямбда-функція?
7. Що таке рекурсія?

4. ОБЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ.

4.1. Основні поняття.

Багато сучасних мов підтримують кілька парадигм програмування (наприклад, директивне, функціональне, об'єктно-орієнтоване). Такі мови є змішаними. До них відносять і Python. Ключову роль в ООП відіграє множина об'єктів. Реальний світ складається з об'єктів та їхньої взаємодії між собою. В результаті взаємодії об'єкти можуть змінюватися самі або змінювати інші об'єкти. У світі умовно можна виділити різні системи, які реалізують певні цілі (змінюються з одного стану в інший). Наприклад, група на занятті – це система, яка складається з таких об'єктів, як діти, учитель, столи, комп'ютери, проектор тощо. У цієї системи можна виділити основну мету – збільшення частки знань дітей на якусь величину. Щоб домогтися цього, об'єкти системи повинні певним чином виконати взаємодію між собою.

Необхідно розуміти різницю між програмою написаною на основі структурного «стилю» і програмою в «стилі» ООП. У першому випадку, на перший план виходить логіка, розуміння послідовності виконання виразів (дій) для досягнення цілей. У другому – важливо системне мислення, вміння бачити систему в цілому, з одного боку, і розуміння ролі її частин (об'єктів), з іншого.

Далі наведено основні принципи, які описують ООП:

1) об'єктно-орієнтована програма складається з об'єктів, які посилають один одному повідомлення;

2) кожний об'єкт може складатися з інших об'єктів (а може і не складатися);

3) кожний об'єкт належить певному класу (типу), який задає поведінку об'єктів, створених на його основі.

Клас – це опис об'єктів певного типу. На основі класів створюють об'єкти. Може існувати множина об'єктів, які належать одному класу. З іншого боку, може існувати клас без об'єктів, реалізованих на його основі. Програма, написана з використанням парадигми ООП, повинна складатися з: об'єктів, класів (опису об'єктів), взаємодій об'єктів між собою, в результаті яких змінюються їх властивості.

Об'єкт в програмі можна створити лише на основі будь-якого класу. Тому, ООП має розпочинатися з проектування і створення класів. Класи можуть бути розташовані або спочатку коду програми, або імпортуватися з інших файлів – модулів (також на початку коду).

Створення класів. Для створення класів передбачена інструкція *class*, яка складається з рядка заголовка і тіла. Заголовок складається з ключового слова *class*, імені класу і, можливо, назв суперкласів в дужках. Суперкласи можуть бути відсутніми, в такому випадку дужки не потрібні. Тіло класу складається з блоку різних інструкцій. Тіло повинно мати відступ (як і будь-які вкладені конструкції в Python). Схематично клас можна подати таким чином:

```
class ІМ‘Я_КЛАСУ:
   змінна = значення ….
   def ІМ‘Я_МЕТОДА(self, ...):
      self.змінна = значення …
```

У заголовку після імені класу можна вказати суперкласи (в дужках), а методи можуть бути більш складними. Методи в класах – це звичайні функції, за одним винятком: вони мають один обов'язковий параметр – *self*, який потрібен для зв'язку з конкретним об'єктом. Атрибути класу – це імена змінних поза функцій і імена функцій. Вони успадковуються всіма об'єктами, створеними на основі даного класу, і забезпечують властивості і поведінку об'єкта. Об'єкти можуть мати атрибути, які створюються в тілі методу, якщо даний метод буде викликано для конкретного об'єкта.

Створення об'єктів. Об'єкти створюються так:

змінна = ІМ‘Я_КЛАСУ()

Після такої інструкції у програмі з'являється об'єкт, доступ до якого можна отримати за ім'ям змінної, пов'язаної з ним. При створенні об'єкт отримує атрибути його класу (він має характеристики, визначені у його класі). Кількість об'єктів, які можна створити на основі певного класу, не обмежена. Об'єкти одного класу мають схожий набір атрибутів, а значення атрибутів можуть бути різними (вони схожі, але

індивідуально різняться).

Self: Методи класу - це невеликі програми, призначені для роботи з об'єктами. Методи можуть створювати нові властивості (дані) об'єкта, змінювати існуючі, виконувати інші дії над об'єктами. Методу необхідно «знати», дані якого об'єкта йому потрібно буде обробляти. Для цього йому в якості першого (а іноді і єдиного) аргументу передається ім'я змінної, пов'язаної з об'єктом. Щоб в описі класу вказати об'єкт, який передається в подальшому, використовують параметр *self*. Виклик методу для конкретного об'єкта в основному блоці програми виглядає таким чином: *ОБ'ЄКТ.ІМ'Я_МЕТОДА(...)*, де *ОБ'ЄКТ* – змінна, пов'язана з ним. Цей вираз перетворюється в класі, до якого відноситься об'єкт, в *ІМ'Я_МЕТОДА(ОБ'ЄКТ, ...)* – замість параметра *self* підставляють конкретний об'єкт. Атрибути екземпляра створюються шляхом присвоювання значень атрибутам об'єкта екземпляра. Зазвичай вони створюються всередині методів класу, в інструкції *class* – присвоєнням значень атрибутам аргументу *self* (який завжди є імовірним екземпляром). Можна створювати атрибути за допомогою операції присвоєння в будь-якому місці програми, де доступне посилання на екземпляр, навіть за межами інструкції *class*.

Конструктор. Зазвичай всі атрибути екземплярів ініціалізують в конструкторі __*init*__. Метод конструктора __*init*__ використовують для встановлення початкових значень атрибутів екземплярів і виконання інших початкових операцій (це - звичайна функція, яка підтримує і можливість визначення значень аргументів за замовчуванням, і передачу іменованих аргументів).

4.2. Абстракція і декомпозиція.

Абстракція в ООП дозволяє скласти з даних і алгоритмів обробки цих даних об'єкти, не беручи до уваги несуттєві (на деякому рівні) з точки зору складеної інформаційної моделі деталей. Таким чином, програма піддається *декомпозиції* на частини "дозованої" складності. Окремий об'єкт, навіть разом з сукупністю його зв'язків з іншими об'єктами, людиною сприймається легше (саме так він звик оперувати в реальному світі), ніж щось неструктуроване і монотонне.

Перед тим як почати написання, навіть самої простої ,об'єктної програми, необхідно провести аналіз предметної області, для того щоб виявити в ній класи об'єктів.

При виділенні об'єктів необхідно абстрагуватися від більшості притаманних їм властивостей і сконцентруватися на властивостях, які є значущими для завдання.

Вдала декомпозиція є трудозатратною. Від неї залежать не тільки кількісні характеристики коду (швидкодія, яку займає пам'ять), але і трудомісткість подальшого розвитку і супроводу.

4.3. Об'єкти.

В даному методичному посібнику *об'єкти* Python зустрічалися багато раз: адже кожне число,рядок, функція, модуль і т.п. - це *об'єкти*. Деякі вбудовані об'єкти мають в Python синтаксичну підтримку (для задання литералов):

```
a = 3
b = 4.0
c = a + b
```

Спочатку ім'я "a" зв'язується в локальному просторі імен з об'єктом-числом 3 (ціле число). Потім "b" зв'язується з об'єктом-числом 4.0 (число з плаваючою точкою). Після цього над об'єктами 3 і 4.0 виконується операція додавання, і ім'я "c" зв'язується з кінцевим об'єктом.

До речі, операціями, в основному, будуть називатися методи, які мають в Python синтаксичну підтримку. Те ж саме можна записати як:

```
c = a.__add__(b)
```

Тут *__add __ ()* - метод об'єкта *a*, який реалізує операцію + між цим та іншим об'єктом.

Дізнатися набір методів для деякого об'єкта можна за допомогою вбудованої функції *dir()*:

```
>>> dir(a)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
'__floordiv__', '__getattribute__', '__getnewargs__', '__hash__',
'__hex__', '__init__', '__int__', '__invert__', '__long__',
'__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
'__nonzero__', '__oct__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__str__', '__sub__', '__truediv__', '__xor__']
```

Тут варто вказати на ще одну особливість Python. Не тільки інфіксні операції, але і вбудовані функції очікують наявності деяких методів у об'єкта:

```
abs(c)
```

Функція abs () насправді використовує метод переданого їй об'єкта:

```
c.__abs__()
```

Об'єкти з'являються в результаті виконання функцій-фабрик або конструкторів класів, а закінчують своє існування при видаленні останнього посилання на об'єкт. Оператор del видаляє ім'я (а значить, і одне посилання на об'єкт) з простору імен:

```
a = 1
# ...
del a
# ім’я більше не існує
```

4.4. Типи і класи.

Тип визначає область допустимих значень об'єкта і набір операцій надним. В ООП тип тісно пов'язаний з поведінкою - діями об'єкта, що складаються в зміні

внутрішнього стану і викликами методів інших об'єктів.

Раніше в мові Python вбудовані типи даних не були екземплярами класу, тому вважалося, що це були просто об'єкти певного типу. Тепер ситуація змінилася, і об'єкти вбудованих типів мають класи, до яких вони належать. Таким чином, тип і клас в Python стають синонімами.

Інтерпретатор мови Python завжди може сказати, до якого типу належить об'єкт. Однак з точки зору застосування об'єкта в операції його приналежність до класу не грає вирішальної ролі: набагато важливіше, які методи підтримує об'єкт.

Екземпляри класів можуть з'являтися в програмі не тільки з литералов або в результаті операцій. Зазвичай для отримання об'єкта класу досить викликати конструктор цього класу з деякими параметрами. Об'єкт-клас, як і об'єкт функції, може бути викликаний. Це і буде викликом конструктора:

```
>>> import sets
>>> s = sets.Set([1, 2, 3])
```

У цьому прикладі модуль sets містить визначення класу Set. Викликається конструктор цього класу з параметром [1, 2, 3]. В результаті з ім'ям s буде зв'язаний об'єкт-множина з трьох елементів 1, 2, 3.

Слід зауважити, що, крім конструктора, певні класи мають і деструктор - метод, який викликається при знищенні об'єкта. В мові Python об'єкт знищується в разі видалення останнього посилання на нього або в результаті сбору сміття, якщо об'єкт виявився в невживаному циклі посилань. Так як Python сам управляє розподілом пам'яті, деструктори в ньому потрібні дуже рідко. Зазвичай в тому випадку, коли об'єкт управляє ресурсом, який потрібно коректно повернути в певний стан.

Ще один спосіб отримати об'єкт деякого типу - використання функцій-фабрик. За синтаксису виклик функції-фабрики не відрізняється від виклику конструктора класу.

Визначення класу

Нехай в ході аналізу даної предметної області необхідно визначити клас Граф. Граф - це безліч вершин і набір ребер, який попарно з'єднує ці вершини. Над графом можна проробляти операції, такі як додавання вершини, ребра, перевірка наявності ребра в графі і т.п. Мовою Python визначення класу може виглядати так:

```
class G:
   def __init__(self, V, E):
      self.vertices = set (V)
      self.edges = set (E)
   def add_vertex(self, v):
      self.vertices.add(v)
   def add_edge(self, v3):
      v1, v2 = v3
      self.vertices.add(v1)
      self.vertices.add(v2)
      self.edges.add((v1, v2))
   def has_edge(self, v3):
      v1, v2 = v3
      return ((v1, v2) in self.edges)
   def __str__(self):
      return ("%s; %s" % (self.vertices, self.edges))
```

Використовувати клас можна наступним чином:

```
g = G([1, 2, 3, 4], [(1, 2), (2, 3), (2, 4)])

print (g)
g.add_vertex(5)
g.add_edge((5,6))
print (g.has_edge((1,6)))
print (g)
```

```
{1, 2, 3, 4}; {(1, 2), (2, 3), (2, 4)}
False
{1, 2, 3, 4, 5, 6}; {(1, 2), (5, 6), (2, 3), (2, 4)}
```

Конструктор класу має спеціальне ім'я *__init__*. (Деструктор тут не потрібен,

але він би мав ім'я *__del__*.) Методи класу визначаються в просторі імен класу. В якості першого формального аргументу методу прийнято використовувати *self*. Крім методів в об'єкті класу є два атрибути: *vertices* (вершини) і *edges* (ребра). Для представлення об'єкту *G* у вигляді строки використовується спеціальний метод *__str__* ().

Належність класу можна з'ясувати за допомогою вбудованої функції *isinstance ():*

print (isinstance (g, G))

4.5. Успадкування, інкапсуляція, поліморфізм.

Ідеї (принципи) ООП. Виділяють такі основні ідеї ООП як успадкування, інкапсуляція і поліморфізм:

1) *Успадкування*. Можливість виділяти загальні властивості і методи класів в один клас верхнього рівня (батьківський). Класи, які мають загального батька, різняться між собою за рахунок включення до їх складу різних додаткових властивостей (атрибутів) і методів.

2) *Інкапсуляція*. Атрибути (властивості) і методи класу класифікують на доступні зовні (опубліковані) і недоступні (захищені). Захищені атрибути можна змінити, перебуваючи поза класом. Опубліковані атрибути також називають інтерфейсом об'єкта, тому що за їх допомогою з об'єктом можна взаємодіяти. Інкапсуляція покликана забезпечити надійність програми, тому що змінити істотні для існування об'єкта атрибути стає неможливо.

3) *Поліморфізм*. Поліморфізм має на увазі заміщення атрибутів, описаних раніше в інших класах: ім'я атрибута залишається колишнім, а реалізація вже іншою. Поліморфізм дозволяє адаптувати класи, залишаючи при цьому одинінтерфейс взаємодії.

Переваги ООП. Серед них можна виділити:

1. Використання одного і того ж програмного коду з різними даними. Класи дозволяють створювати множину об'єктів, кожен з яких має власні значення атрибутів. Немає потреби вводити множину змінних (об'єкти отримують в своє

розпорядження індивідуальний простір імен). Простір імен конкретного об'єкта формується на основі класу, від якого він був створений, а також на основі усіх батьківських класів даного класу.

2. Успадкування і поліморфізм дозволяють не писати новий код, а налаштовувати вже існуючий, за рахунок додавання і перевизначення атрибутів. Це веде до скорочення обсягу вихідного коду. ООП дозволяє скоротити час на написання вихідного коду, проте ООП завжди передбачає велику роль попереднього аналізу предметної області, попереднього проектування. Від правильності розв‘язків на цьому попередньому етапі залежить куди більше, ніж від безпосереднього написання вихідного коду.

Особливості ООП в Python. У порівнянні з іншими поширеними мовами програмування у Python можна виділити такі особливості, пов'язані з ООП:

1) *Будь-які дані (значення) – це об'єкти*. Число, рядок, список, масив і ін.– все є об'єктом. Бувають об'єкти вбудованих класів (ті, що перераховані в попередньому реченні), а бувають об'єкти класів користувача (їх створює програміст). Для єдиного механізму взаємодії передбачені методи перезавантаження операторів .

2) *Клас – це об'єкт з власним простором імен*. Тому правильніше було вживати замість слова «об'єкт», слово «екземпляр». І говорити «екземпляр об'єкта», маючи під цим на увазі створений на основі класу саме об'єкт, і «екземпляр класу», маючи на увазі сам клас як об'єкт.

3) *Інкапсуляції в Python не приділяється особливої уваги*. В інших мовах програмування зазвичай не можна отримати безпосередньо доступ до властивості, описаного в класі. Для його зміни може бути передбачений спеціальний метод. В Python це легко зробити, звернувшись до властивості класу із зовні. Незважаючи на це в Python передбачені спеціальні способи обмеження доступу до змінних в класі.

ПРАКТИЧНА РОБОТА

1) Створення екземпляра класу

```
# Оголошення порожнього класу MyClass
class MyClass: pass
obj = MyClass()
# Объект obj - це екземпляр класу MyClass,
# (він має тип MyClass)
print(type(obj)) # <class '__main__.MyClass'>
# MyClass – це клас, він є об'єктом, екземпляром метакласу type
# який є абстракцією поняття типу даних
print(type(MyClass)) # <class 'type'>
# Тому з класами можна виконувати операції як із об'єктами наприклад, копиювання
AnotherClass = MyClass
print(type(AnotherClass))
# тепер AnotherClass – це те ж саме, що і MyClass,
# і obj є екземпляром класу AnotherClass
print(isinstance(obj, AnotherClass)) # True
```

2) Всі елементи клада називають атрибутами. Оголошення класу *MyClass* з двома атрибутами *int_field, str_field*, які є змінними

```
class MyClass:
    int_field = 8
    str_field = 'a string'
# Звернення до атрибутів класу
print (MyClass.int_field); print (MyClass.str_field)
# Створення двох екземплярів класу
object1 = MyClass (); object2 = MyClass ()
# Звернення до атрибутів класу через його екземпляри
print (object1.int_field); print (object2.str_field)
# Всі перераховані вище звернення до атрибутів насправді відносяться
```

 # до двох одних і тих самих змінних

 # Зміна значення атрибута класу

MyClass.int_field = 10

print (MyClass.int_field); print (object1.int_field); print(object2.int_field)

 # Однак, аналогічно до глобальних і локальних змінних,

 # присвоєння значення атрибуту об'єкта не змінює значення

 # атрибута класу, а веде до створення атрибута даних (нестатичного поля)

object1.str_field = 'another string'

print(MyClass.str_field); print(object1.str_field);

print(object2.str_field)

3) Атрибути-дані аналогічні полям. Їх не треба описувати: як і змінні, вони створюються в момент першого присвоювання.

 # Клас, який описує людину

class Person: pass

 # Створення екземплярів класу

alex = Person()

alex.name = 'Alex'; alex.age = 18

john = Person()

john.name = 'John'; john.age = 20

 # Атрибути-дані відносять тільки до окремих екземплярів класу

 # і ніяк не впливають на значення відповідних атрибутів-даних інших екземплярів

print(alex.name, 'is', alex.age); print(john.name, 'is', john.age)

4) Атрибутами класу можуть бути й методи-функції

 # Клас, який описує людину

```
class Person:
# Перший аргумент, який вказує на поточний екземпляр класу,
# прийнято називати self
   def print_info(self):
      print(self.name, 'is', self.age)
# Створення екземплярів класу
alex=Person(); alex.name='Alex'; alex.age=18
john = Person(); john.name = 'John'; john.age = 20
# Перевіримо, чим є атрибут-функція print_info класу Person
print(type(Person.print_info)) # функція (<class 'function'>)
# Викличемо його для об‘єктів alex і john
Person.print_info(alex)
Person.print_info(john)
# Метод – функція, зв‘язана з об‘єктом. Всі атрибути класу, які є
# функціями, описують відповідні методи екземплярів даного класу
print(type(alex.print_info)) # метод (<class 'method'>)
# Виклик методу print_info
alex.print_info()
john.print_info()
```

5) Створимо клас, значення початкових атрибутів (з методу __init__) якого залежить від переданих аргументів при створенні об'єктів. Далі ці атрибути об'єктів, створених на основі даного класу, можна змінювати за допомогою методів.

```
class Building:
   def __init__(self,w,c,n=0):
      self.what=w; self.color=c; self.numbers=n; self.mwhere(n)
   def mwhere(self,n):
      if n <= 0: self.where = "відсутні"
```

```
        elif 0 < n < 100: self.where = "малий склад"
        else: self.where = "основний склад"
    def plus(self,p):
        self.numbers = self.numbers + p
        self.mwhere(self.numbers)
    def minus(self,m):
        self.numbers=self.numbers-m
        self.mwhere(self.numbers)


m1 = Building("дошки", "білі",50)
m2 = Building("дошки", "коричневі", 300)
m3 = Building("цегла","біла")
print (m1.what,m1.color,m1.where)
print (m2.what,m2.color,m2.where)
print (m3.what,m3.color,m3.where)
m1.plus(500);print (m1.numbers, m1.where)
```

Висновок

В даному розділі були розглянуті основи об’єктно-орієнтованої парадигми програмування в мові програмування Python 3. Пройшовши даний розділ студент зможе:

1) визначати клас/об’єкт та використовувати їх в своїй подальшій роботі;
2) створювати атрибути та методи для відповідних класів/об’єктів;
3) вміти користуватись конструктором та деструктором.

Питання для самоконтролю

1. Що таке клас? Як його створити в Python-програмі?
2. Що таке об’єкт? Як його створити в Python-програмі?
3. Що таке атрибут/метод?
4. Дати визначення абстракції та декомпозиції.
5. Дати визначення типу в ООП.
6. Що таке конструктор?
7. Дати визначення наступним поняттям:
    1) успадкування;
    2) інкапсуляція;
    3) поліморфізм.

5. ЧИСЕЛЬНІ АЛГОРИТМИ. МАТРИЧНІ ОБЧИСЛЕННЯ.

5.1. Модуль Numpy.

Numpy — розширення мови Python, що додає підтримку великих багатовимірних масивів і матриць, разом з великою бібліотекою високорівневих математичних функцій для операцій з цими масивами.

Оскільки Python — інтерпретована мова, математичні алгоритми, часто працюють в ньому набагато повільніше ніж у компільованих мовах, таких як C або навіть Java. NumPy намагається вирішити цю проблему для великої кількості обчислювальних алгоритмів забезпечуючи підтримку багатовимірних масивів і безліч функцій і операторів для роботи з ними. Таким чином будь-який алгоритм який може бути виражений в основному як послідовність операцій над масивами і матрицями працює також швидко як еквівалентний код написаний на C.

Для того, щоб встановити NumPy, слід:

а) Linux:

sudo pip3 install numpy

б) Windows:

pip3 install numpy

5.2. Створення масиву.

Основним об'єктом *NumPy* є однорідний багатовимірний масив (в *numpy* називається *numpy.ndarray*). Це багатомірний масив елементів (зазвичай чисел), одного типу.

Найбільш важливі атрибути об'єктів *ndarray*:

ndarray.ndim - число вимірювань (частіше їх називають "осі") масиву.

ndarray.shape - розміри масиву, його форма. Це кортеж натуральних чисел, що показує довжину масиву по кожній осі. Для матриці з n рядків і m стовпців, *shape* буде (n, m). Число елементів кортежу *shape* дорівнює *ndim*.

ndarray.size - кількість елементів масиву. Очевидно, дорівнює добутку всіх елементів атрибута *shape*.

ndarray.dtype - об'єкт, що описує тип елементів масиву.

ndarray.itemsize - розмір кожного елемента масиву в байтах.

ndarray.data - буфер, який містить фактичні елементи масиву.

У NumPy існує багато способів створити масив. Один з найбільш простих - створити масив із звичайних списків або кортежів Python, використовуючи функцію *numpy.array()* (функція, що створює об'єкт типу ndarray):

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

Функція *array()* трансформує вкладені послідовності в багатовимірні масиви. Тип елементів масиву залежить від типу елементів початкової послідовності (але також можна і перевизначити його в момент створення).

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Також можна змінити тип в момент створення:

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

Функція *array()* не єдина функція для створення масивів. Зазвичай елементи масиву спочатку невідомі, а масив, в якому вони будуть зберігатися, вже потрібен. Тому є кілька функцій для того, щоб створювати масиви з якимось вихідним вмістом (за замовчуванням тип створюваного масиву - *float64*).

Функція *zeros()* створює масив з нулів, а функція *ones()* - масив з одиниць. Обидві функції приймають кортеж з розмірами, і аргумент *dtype*:

```
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функція *eye()* створює одиничну матрицю (двомірний масив):

```
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Функція empty () створює масив без його заповнення. Початковий вміст формується випадково і залежить від стану пам'яті на момент створення масиву (тобто від того сміття, що в ній зберігається):

```
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920149e-310],
       [  6.93920058e-310,   6.93920058e-310,   6.93920058e-310],
       [  6.93920359e-310,   0.00000000e+000,   6.93920501e-310]])
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920147e-310],
       [  6.93920149e-310,   6.93920146e-310,   6.93920359e-310],
       [  6.93920359e-310,   0.00000000e+000,   3.95252517e-322]])
```

Для створення послідовностей чисел, в NumPy є функція *arange()*, аналогічна вбудованої в Python *range()*, тільки замість списків вона повертає масиви, і приймає не тільки цілі значення:

```
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 1, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

Взагалі, при використанні *arange()* з аргументами типу *float*, складно бути впевненим в тому, скільки елементів буде отримано (через обмеження точності чисел з плаваючою комою). Тому, в таких випадках зазвичай краще використовувати функцію *linspace()*, яка замість кроку в якості одного з аргументів приймає число, що дорівнює кількості потрібних елементів:

```
>>> np.linspace(0, 2, 9)  # 9 чисел від 0 до 2 включно
array([  0. ,  0.25,  0.5 ,  0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

fromfunction(): застосовує функцію до всіх комбінацій індексів

```
>>> def f1(i, j):
...     return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0.,  1.,  2.,  3.],
       [ 3.,  4.,  5.,  6.],
       [ 6.,  7.,  8.,  9.]])
>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

Якщо масив занадто великий, щоб його друкувати, NumPy автоматично приховує центральну частину масиву і виводить тільки його крайні значення.

```
>>> print(np.arange(0, 3000, 1))
[   0    1    2 ..., 2997 2998 2999]
```

Якщо вам дійсно потрібно вивести весь масив, використовуйте функцію *numpy.set_printoptions*:

```
np.set_printoptions(threshold=np.nan)
```

5.3. Функції для роботи з масивами.

Математичні операції над масивами виконуються поелементно. Створюється новий масив, який заповнюється результатами дії оператора.

```
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([  0,  30,  80, 150])
>>> a / b
array([        inf,  30.        ,  20.        ,  16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in true_divide
>>> a ** b
array([     1,     30,   1600, 125000])
>>> a % b
<string>:1: RuntimeWarning: divide by zero encountered in remainder
array([0, 0, 0, 2])
```

Для цього, звісно, масиви повинні бути однакових розмірів.

```
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> d = np.array([[1, 2], [3, 4], [5, 6]])
>>> c + d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

Також можна робити математичні операції між масивом і числом. В цьому випадку між заданим числом та кожним елементом проводиться задана маніпуляція.

```
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
```

```
array([  8000,  27000,  64000, 125000])
>>> a < 35
array([ True,  True, False, False], dtype=bool)
```

NumPy також надає безліч математичних операцій для обробки масивів (Див. https://docs.scipy.org/doc/numpy/reference/routines.math.html):

```
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,  0.96496603])
>>> np.arctan(a)
array([ 1.52083793,  1.53747533,  1.54580153,  1.55079899])
>>> np.sinh(a)
array([  2.42582598e+08,   5.34323729e+12,   1.17692633e+17,
         2.59235276e+21])
```

Багато унарних операцій, такі як, наприклад, обчислення суми всіх елементів масиву, представлені також і у вигляді методів класу *ndarray*.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

За замовчуванням, дані операції застосовуються до масиву так, якби він був списком чисел, незалежно від його форми. Однак, вказавши параметр *axis*, можна застосувати операцію для зазначеної осі масиву:

```
>>> a.min(axis=0)  # Найменше число в кожному стовпці
array([1, 2, 3])
>>> a.min(axis=1)  # Найменше число в кожному рядку
array([1, 4])
```

5.4.Індекси, зрізи, ітерації.

Одномірні масиви здійснюють операції індексування, зрізу та ітерацій, що дуже схожі на звичайні операції зі списками і іншими послідовностями Python (видаляти за допомогою зрізів не можна).

```
>>> a = np.arange(10) ** 3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[1]
1
>>> a[3:7]
array([ 27,  64, 125, 216])
>>> a[3:7] = 8
>>> a
array([  0,   1,   8,   8,   8,   8,   8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343,   8,   8,   8,   8,   8,   1,   0])
>>> del a[4:6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: cannot delete array elements
>>> for i in a:
...     print(i ** (1/3))
...
0.0
1.0
2.0
2.0
2.0
2.0
2.0
7.0
8.0
9.0
```

У багатовимірних масивів на кожну вісь припадає один індекс. Індекси передаються у вигляді послідовності чисел, розділених комами (тобто кортежами):

```
>>> b = np.array([[  0, 1, 2, 3],
...              [10, 11, 12, 13],
...              [20, 21, 22, 23],
...              [30, 31, 32, 33],
...              [40, 41, 42, 43]])
...
>>> b[2,3]  # Друга строка, третій стовпець
23
>>> b[(2,3)]
23
>>> b[2][3]  # або так
23
>>> b[:,2]  # третій стовпець
array([ 2, 12, 22, 32, 42])
>>> b[:2]  # Перші дві строки
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
>>> b[1:3, : : ]  # Друга та третя строки
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Коли індексів менше, ніж осей, відсутні індекси передбачаються доповненими за допомогою зрізів:

```
>>> b[-1]  # Остання строка. Еквівалентно b[-1,:]
array([40, 41, 42, 43])
```

b [i] можна читати як b [i, <стільки символів ':', скільки потрібно>]. У NumPy це також може бути записано за допомогою крапок, як b [i, ...].

Наприклад, якщо x має ранг 5 (тобто у нього 5 осей), тоді

```
x [1, 2, ...] еквівалентно x [1, 2,:,:,:],
x [..., 3] те ж саме, що x [:,:,:,:, 3] і
x [4, ..., 5,:] це x [4,:,:, 5,:].
```

```
>>> a = np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102], [110, 112, 113]]))
>>> a.shape
(2, 2, 3)
```

```
>>> a[1, ...]  # теж саме, що і a[1, : , :] або a[1]
array([[100, 101, 102],
      [110, 112, 113]])
>>> c[... ,2]  # теж саме, що і a[: , : ,2]
array([[  2,  13],
      [102, 113]])
```

Ітерування багатомірних масивів починається з першої осі:

```
>>> for row in a:
...     print(row)
...
[[ 0  1  2]
 [10 12 13]]
[[100 101 102]
 [110 112 113]]
```

Проте, якщо потрібно перебрати поелементно весь масив так, якщо б він був одновимірним, для цього можна використовувати атрибут *flat*:

```
>>> for el in a.flat:
...     print(el)
...
0
1
2
10
12
13
100
101
102
110
112
113
```

5.5. Функції модуля Numpy.

5.5.1. Маніпуляції з формою

Кожен масив має форму (*shape*), яка визначається числом елементів вздовж кожної осі:

```
>>> a
array([[[  0,   1,   2],
        [ 10,  12,  13]],

       [[100, 101, 102],
        [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

Форма масиву може бути змінена за допомогою різних команд:

```
>>> a.ravel()  # Робить масив плоским
array([  0,   1,   2,  10,  12,  13, 100, 101, 102, 110, 112, 113])
>>> a.shape = (6, 2)  # зміна форми
>>> a
array([[  0,   1],
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.transpose()  # транспонування
array([[  0,   2,  12, 100, 102, 112],
       [  1,  10,  13, 101, 110, 113]])
>>> a.reshape((3, 4))  # зміна форми
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

Порядок елементів в масиві в результаті функції *ravel()* відповідає звичайному "C-стилю", тобто, чим правіше індекс, тим він "швидше змінюється": за елементом a [0,0] буде йти a [0,1].

```
>>> a
array([[  0,   1],
```

```
       [  2,  10],
       [ 12,  13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.reshape((3, 4), order='F')
array([[  0, 100,   1, 101],
       [  2, 102,  10, 110],
       [ 12, 112,  13, 113]])
```

Метод *reshape()* повертає її аргумент зі зміненою формою, в той час як метод *resize()* змінює сам масив:

```
>>> a.resize((2, 6))
>>> a
array([[  0,   1,   2,  10,  12,  13],
       [100, 101, 102, 110, 112, 113]])
```

Якщо при операції такої перебудови один з аргументів задається як -1, то він автоматично розраховується відповідно з іншими заданими:

```
>>> a.reshape((3, -1))
array([[  0,   1,   2,  10],
       [ 12,  13, 100, 101],
       [102, 110, 112, 113]])
```

5.5.2. Об'єднання масивів

Кілька масивів можуть бути об'єднані разом вздовж різних вісей за допомогою функцій *hstack* і *vstack*.

hstack() об'єднує масиви за першими вісями, *vstack()* - за останніми:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
```

```
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Функція *column_stack()* об'єднує одномірні масиви в якості стовпців двовимірного масиву:

```
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Аналогічно для рядків є функція *row_stack()*.

```
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

5.5.3. Розбиття масиву

Використовуючи *hsplit()* можна розбити масив вздовж горизонтальної осі, вказавши або число повернутих масивів однакової форми, або номери стовпців, після яких масив розрізається:

```
>>> a = np.arange(12).reshape((2, 6))
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3)  # Разбити на 3 частини
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]]),
 array([[ 4,  5], [10, 11]])]
>>> np.hsplit(a, (3, 4))  # Розрізати a після третього і четвертого стовпця
[array([[0, 1, 2], [6, 7, 8]]),
 array([[3], [9]]),
 array([[ 4,  5], [10, 11]])]
```

Функція *vsplit()* розбиває масив вздовж вертикальної осі, а *array_split()* дозволяє вказати осі, вздовж яких відбудеться розбиття.

ПРАКТИЧНА РОБОТА

1) Створити вектор розміром 10, заповнений нулями, але п'ятий елемент дорівнює 1

```
import numpy as np
Z = np.zeros(10)
Z[4] = 1
print(Z)
```

2) Створити вектор зі значеннями від 10 до 49

```
import numpy as np
Z = np.arange(10,50)
print(Z)
```

3) Розвернути вектор (перший елемент стає останнім)

```
import numpy as np
Z = np.arange(50)
Z = Z[::-1]
print(Z)
```

4) Створити матрицю (двомірний масив) 3x3 зі значеннями від 0 до 8

```
import numpy as np
Z = np.arange(9).reshape(3,3)
print(Z)
```

5) Створити масив 10x10 з випадковими значеннями, знайти мінімум і максимум

```
import numpy as np
Z = np.random.random((10,10))
Zmin, Zmax = Z.min(), Z.max()
print(Zmin, Zmax)
```

6) Створити випадковий вектор розміром 30 і знайти середнє значення всіх елементів

```
import numpy as np
Z = np.random.random(30)
m = Z.mean()
print(m)
```

7) Створити 8x8 матрицю і заповнити її в шаховому порядку

```
import numpy as np
Z = np.zeros((8,8), dtype=int)
Z[1::2,::2] = 1
Z[::2,1::2] = 1
print(Z)
```

8) Дано масив 10x2 (точки в декартовій системі координат), перетворити в полярну

```
import numpy as np
Z = np.random.random((10,2))
X,Y = Z[:,0], Z[:,1]
R = np.hypot(X, Y)
T = np.arctan2(Y,X)
print(R)
print(T)
```

9) Замінити максимальний елемент на нуль

```
import numpy as np
Z = np.random.random(10)
Z[Z.argmax()] = 0
print(Z)
```

10) Знайти найближче до заданого значення число в заданому масиві

```
import numpy as np
Z = np.arange(100)
v = np.random.uniform(0,100)
index = (np.abs(Z-v)).argmin()
print(Z[index])
```

Висновок

В даному розділі була розглянута математична бібліотека Numpy, яка використовується в мові програмування Python 3. Пройшовши даний розділ студент ознайомиться з основними методами та властивостями масивів в Python 3, а також зможе проводити складні математичні обчислення.

Питання для самоконтролю

1. Що таке Numpy? Для яких цілей вона використовується.
2. Як створити масив за допомогою Numpy? Назвіть основні атрибути масиву.
3. Назвіть основні функції роботи з масивами.
4. Що таке індекс в масиві? Який індекс повинен бути в першого елемента масиву?
5. Що таке зріз в масиві? Наведіть приклад зрізу.
6. Що таке “форма” в Numpy? Наведіть приклад форми.
7. Як об’єднати декілька масивів в один? Як розбити масив? Наведіть приклад.

6. ОБРОБКА ТЕКСТІВ.

6.1.Рядки.

Рядки в мові Python є типом даних, спеціально призначеним для обробки текстової інформації. Рядок може містити текст довільної довжини, який обмежено наявною пам'яттю ПК.

У нових версіях Python є два типи рядків: звичайні рядки (послідовність байтів) і Unicode-рядка (послідовність символів). В Unicode-рядку кожен символ може займати в пам'яті 2 або 4 байти, в залежності від налаштувань періоду компіляції. Чотирьохбайтові знаки використовуються в основному для східних мов.

6.2.Кодування python-програми.

Для того, щоб Unicode-літерали в Python-програмі сприймалися інтерпретатором правильно, необхідно вказати кодування на початку програми, записавши в першому або другому рядку приблизно таке (для Unix/Linux):

 # -*- coding: koi8-r -*-
або (під Windows):
 # -*- coding: cp1251 -*-

Можуть бути й інші варіанти:

 # -*- coding: latin-1 -*-
 # -*- coding: utf-8 -*-
 # -*- coding: mac-cyrillic -*-
 # -*- coding: iso8859-5 -*-
Повний перелік кодувань (та їхніх псевдонімів):

```
import encodings.aliases
print( encodings.aliases.aliases)
```

```
{'646': 'ascii', 'ansi_x3.4_1968': 'ascii', 'ansi_x3_4_1968': 'ascii',
```

'ansi_x3.4_1986': 'ascii', 'cp367': 'ascii', 'csascii': 'ascii', 'ibm367': 'ascii', 'iso646_us': 'ascii', 'iso_646.irv_1991': 'ascii', 'iso_ir_6': 'ascii', 'us': 'ascii', 'us_ascii': 'ascii', 'base64': 'base64_codec', 'base_64': 'base64_codec', 'big5_tw': 'big5', 'csbig5': 'big5', 'big5_hkscs': 'big5hkscs', 'hkscs': 'big5hkscs', 'bz2': 'bz2_codec', …}

Якщо кодування не вказано, то вважається, що використовується us-ASCII, яке зберігає символи англійської мови, як числа що знаходяться між 0 та 127. (65 велике “A”, 97 маленьке “a”, і т.п.).

6.3.Рядкові літерали.

Python дуже багатий на операції з рядковими об'єктами. Рядки можна визначити у програмі за допомогою рядкових літералів. Літерали записуються з використанням апострофів ', лапок " або цих самих символів, взятих тричі. Всередині літералів зворотна коса риска має спеціальне значення. Вона служить для введення спеціальних символів та для визначення символів через коди. Якщо перед рядковим літералом поставлено r, зворотна коса риска не має спеціального значення (r від англійського слова raw, рядок вказуєтья "як є"). Unicode-літерали вказуються із префіксом u. Наведемо кілька прикладів:

```
s1 = "рядок 1"
s2 = r'\1\2'
s3 = """apple\ntree"""# \n - символ переведення рядка
s4 = """apple
tree"""# рядок у потроєних лапках може мати всередині переведення рядків
s5 = '\x73\65'
u1 = u"Unicode literal"
u2 = u'\u0410\u0434\u0440\u0435\u0441'
```

Зворотна коса риска не повинна бути останнім символом у літералі, тобто, "str\" викличе синтаксичну помилку. Вказування кодування дозволяє використовувати в Unicode-літералах наведений на початку програми тип кодування. Якщо тип кодування не вказано, можна користуватися тільки кодами символів, визначеними через зворотну косу риску.

6.4.Операції над рядками.

До операцій над рядками, які мають спеціальну синтаксичну підтримку в мові, відносяться, зокрема конкатенація (склеювання) рядків, повторення рядка, форматування:

```
>>> print ("A" + "B", "A"*5, "%s" % "A")
AB AAAAA A
```

В операції форматування лівий операнд є рядком формату, а правий може бути або кортежем, або словником, або деяким значенням іншого типу:

```
print( "%i" % 234)
#234
print ("%i %s %3.2f" % (5, "ABC", 23.45678))
#5 ABC 23.46
a = 123
b = [1, 2, 3]
print( "%(a)i: %(b)s" % vars())
#123: [1, 2, 3]
```

6.5.Модуль UNICOD.

Починаючи з версії 2.0, Python має новий тип даних для зберігання тексту: юнікодовий об'єкт. Він може використовуватися для зберігання та обробки даних у Unicode і добре інтегрується з існуючими рядковими об'єктами та здійснює автоматичну конверсію за потребою.

Перевага кодування Unicode полягає в тому, що воно визначає єдину ординальну величину для кожного символа у будь-якій письмовій системі сучасних чи давніх мов. До цього існувало лише 256 ординальних величин для позначення символів і текст здебільшого прив'язувався до кодування, що поєднувало ординальні величини з символами алфавіту. Це призводило до численних непорозумінь, особливо при інтернаціоналізації (цей термін традиційно позначається як "i18n" -- "i" + 18 символів посередині + "n") програмного забезпечення. Ця проблема

вирішена в кодуванні Unicode, де одна кодова сторінка описує всі скрипти.

Створення юнікодових рядків у Python таке ж просте, як і створення звичайних рядків:

```
>>> u'Hello World !'
'Hello World !'
```

Маленький символ *"u"* перед лапками означає, що задається юнікодовий рядок. Задання спеціальних символів у рядку може бути зроблено за допомогою юнікодових контрольних послідовностей (Unicode-Escape encoding) мови Python, як це показано у наступному прикладі:

```
>>> u'Hello\u0020World !'
'Hello World !'
```

Контрольна послідовність *\u0020* вказує, що у заданій позиції повинен бути вставлений символ, що має ординальну величину *0x0020* (пробіл).

Інші символи інтерпретуються через пряме використання їхніх ординальних величин як ординальних величин кодування Unicode. Якщо ваші буквальні величини задано за допомогою кодування Latin-1, що використовується у багатьох західних країнах, то для вас перші 256 символів кодування Unicode ті самі, що й 256 символів кодування Latin-1.

Окрім цих стандартних кодувань, Python має багато інших способів для створення юнікодових рядків на основі певного відомого кодування.

Вбудована функція unicode() надає доступ до всіх зареєстрованих юнікодових кодеків (codec < "COders and DECoders"). Серед найвідоміших кодувань, що можуть конвертуватися цими кодеками - Latin-1, ASCII, UTF-8, та UTF-16. Останні два — кодування змінної довжини, що зберігають юнікодові символи в одному чи більше байтах. Типове кодування — це здебільшого ASCII, що дозволяє лише симлоли від 0 до 127 і видає помилку, коли знаходить інші символи. Коли юнікодовий рядок виводиться на стандартний вивід, записується у файл чи конвертується за допомогою *str()*, то конверсія відбувається за допомогою цього стандартного кодування:

```
>>> u"abc"
'abc'
>>> str(u"abc")
```

```
'abc'
>>> u"&auml;&ouml;&uuml;"
'&auml;&ouml;&uuml;'
>>> str(u"&auml;&ouml;&uuml;")
'&auml;&ouml;&uuml;'
```

6.6.Методи рядків.

Таблиця 6.1 – методи рядків

Метод	Опис
center(w)	Центрує рядок у поле завдовжки *w*
count(sub)	Повертає кількість входжень рядка *sub* у рядок
encode([enc[, errors]])	Повертає рядок у кодуванні *enc*. Параметр *errors* може набувати значення "*strict*" (за замовчуванням), "*ignore*", "*replace*" або "*xmlcharrefreplace*"
endswith(suffix)	Чи закінчується рядок на *suffix*?
expandtabs([tabsize])	Заміняє символи табуляції на пробіли. За замовчуванням *tabsize*=8
find(sub [,start [,end]])	Повертає найменший індекс, із якого починається входження підрядка *sub* у рядок. Параметри *start* та *end* обмежують пошук вікном *start:end*, але повертається індекс, що відповідає вихідному рядку. Якщо підрядок не знайдено, повертається –1
index(sub[, start[, end]])	Аналогічно **find**(), але генерує виняткову ситуацію **ValueError** у разі невдачі
alnum()	Повертає *True*, якщо рядок містить тільки літери та цифри, і має ненульову довжину.Інакше – *False*
Isalpha()	Повертає *True*, якщо рядок містить тільки букви та має ненульову довжину
isdecimal()	Повертає *True*, якщо рядок містить тільки десяткові знаки (тільки для рядків Unicode) та має ненульову довжину
isdigit()	Повертає *True*, якщо містить тільки цифри та має ненульову довжину
islower()	Повертає *True*, якщо всі букви малі (і їх більше однієї), інакше – *False*
isnumeric()	Повертає *True*, якщо в рядку лише числові знаки (тільки для Unicode)
isspace()	Повертає *True*, якщо рядок складається тільки із символів пробілу. **Увага!** Для порожнього рядка повертається *False*
join(seq)	З'єднання рядків із послідовності *seq* через роздільник, указаний рядком
lower()	Робить усі літери в рядку малими

Таблиця 6.1(продовження) – методи рядків

lstrip()	Видаляє символи пробілу ліворуч
replace(old, new[, n])	Повертає копію рядка, у якому підрядки *old* замінено на *new*. Якщо вказано параметр *n*, то заміняються тільки перші *n* входжень
rstrip()	Видаляє символи пробілу праворуч
split([sep[, n]])	Повертає список підрядків, що отримуються розбиттям рядка *a* роздільником *sep*. Параметр *n* визначає максимальну кількість розбиттів (ліворуч)
startswith(prefix)	Чи починається рядок із підрядка *prefix*?
strip()	Видаляє пробільні символи на початку й у кінці рядка
upper()	Робить усі літери в рядку великими

У наступному прикладі використовуються методи *split()* та *join()* для розбиття рядка у список (по роздільниках) та повторне об'єднання списку рядків у рядок:

```
s = "This is an example."
lst = s.split(" ")
print( lst)
#['This', 'is', 'an', 'example.']
s2 = "\n".join(lst)
print (s2)
#This
#is
#an
#example.
```

Щоб перевірити чи закінчується рядок певним сполученням літер, можна використати метод *endswith()*:

```
filenames = ["file.txt", "image.jpg", "str.txt"]
for fn in filenames:
    if fn.lower().endswith(".txt"):
        print(fn)
```

file.txt
str.txt

Шукати в рядку можна за методом *find()*. Наступна програма виведе перше входження символу *e* в рядку *st* починаючи із 7 позиції:

```
st = "Hello world. I'm a python developer"
print(st.find("e", 6))
27
```

Важливим для перетворення текстової інформації є метод *replace():*

```
>>> a = "Це текст , у якому є неправильно поставлені коми"
>>> b = a.replace(" ,", ",")
>>> print(b)
Це текст, у якому є неправильно поставлені коми
```

ПРАКТИЧНА РОБОТА

1) Використання рядків.

А)

```
s1 = "Рядок 1";s2 = 'Рядок 2'
print(s1, s2)
# формування рядка з іншого значення
s3 = str(8); print(s3)
# рядки, які складаються з багатьох рядків
s4 = """ Lesson2. Variables and Data Types
Some data types explained in this lesson:
- int, - bool, - float, - complex, - str """
print(s4)
# символ \ використовують, щоб продовжити рядок
# або будь-який вираз в Python з наступного рядка кода
s5 = "started\
continued"
print(s5)
```

Б)

```
string = "a string" # Створення рядка
# Виведення окремих символів рядка
print(string[0]) # 'a'
print(string[2]) # 's'
print(string[-1]) # 'g'
```

В)

```
string = "a string" # Створення рядка
# Виведення зрізів (групи символів) рядка
print(string[2:5]) # str
print(string[:5]) # a str
print(string[2:]) # string
print(string[::2]) # asrn
# Отримання окремих елементів рядка та їх конкатенація
print(string[2] + string[-3:]) # sing
```

Г)

```
string = input('Введіть рядок: ') # Введення рядка
# Перевірка, чи є в даному рядку символ «q»
if 'q' in string:
   print('В цьому рядку є символ "q"')
else:
   print('В цьому рядку немає символу "q"')
```

Д)

```
string = input('Введіть рядок: ') # Введення рядка
# Виведення довжини рядка
print(‘Довжина цього рядка:’, len(string))
```

2) Приклади операцій над рядками

```
str1 = 'hel'; str2 = 'lo'
result = str1 + str2 # конкатенація рядків
print(result)
# форматування рядків
a = 48;b = 73
message1 = '%d + %d = %d' %(a, b, a + b)
print(message1)
```

```
message2 = '{} - {} = {}'.format(a, b, a - b)
print(message2)
# індексація рядків
s = 'Hello, World!'
print(s[0]) # індексація розпочинається з нуля
print(s[4]) # четвертий (п‘ятий реально) елемент (символ)
print(s[-1]) # від‘ємні числа – індексація розпочинається з кінця
print(s[2:7])
# - символи з 2 (включно) по 7 (не включно)
print(s[2:7:2]) # теж саме, але з кроком два
```

3) Вводиться ціле число N (від 1 до 9), а виводяться рядки з числами, які утворюють визначений «рисунок»

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

```
try:
   N=int(input('Введіть N = '))
except Exception:
   print('Введіть число!!!')
else:
   M=N; pp=''
   while M!=0:
      i=M; L=[]
      while i!=0:
         if i<=M:
            L.append(str(i)); i-=1
      a=list(L)
      pp=''.join(a)
      print(pp); M=M-1
```

4) Визначити середнє арифметичне заданої непустої послідовності додатних цілих чисел, за якою слідує «0» (це ознака кінця послідовності).

```
from math import *
a=input ('Input first number: ')
if not a.isdigit():
   print("String value can not be entered")
   print("or number is negative, restart the program!")
   exit ()
else:
   a=int(a)
if a==0:
   print("The number can not be zero"); input ()

count=0;ar=0
while True:
   ar+=a; count+=1
   try:
      a=int(input('Input next number or Enter 0 to finish: '))
   except:
      print("String value can not be entered")
      print("or number is negative, restart the program!")
      exit ()
   else:
      if a==0: break

ar=ar/count; print("Average: ",ar)
```

Висновок

В даному розділі було розглянуто строковий тип даних в мові програмування Python 3. Пройшовши даний розділ студент ознайомиться з основними методами та властивостями рядків в Python 3.

Питання для самоконтролю

1. Дайте визначення рядку в мові Python.
2. Яким чином можна задати кодування символів в Python-модулі?
3. Що таке рядкові літерали? Наведіть приклади рядкових літералів.
4. Наведіть приклади операцій над рядками.
5. Дайте визначення Unicode.
6. Наведіть приклади рядкових методів.

7. РОБОТА З ДАНИМИ В РІЗНИХ ФОРМАТАХ.

7.1. XML.

XML (Extensible Markup Language, розширювана мова розмітки) дозволяє налагоджувати взаємодію між програмами різних виробників, зберігати та обробляти складноструктуровані дані.

Мова XML (як і html) є підмножиною SGML, але її застосування не обмежені системою www. У XML можна створювати власні набори тегів для конкретної предметної області. У XML можна зберігати й обробляти бази даних та знань, протоколи взаємодії між об'єктами, опису ресурсів і багато чого іншого.

Перевага XML полягає в тому, що разом із даними вона зберігає й контекстну інформацію: теги та їхні атрибути мають імена.

Для подання букв та інших символів XML використовує Unicode, що зменшує проблеми з поданням символів різних алфавітів.

Наступний приклад достатньо простого XML-документа дає уявлення про цей формат (файл expression.xml).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<expression>
<operation type="+">
<operand>2</operand>
<operand>
<operation type="*">
<operand>3</operand>
<operand>4</operand>
</operation>
</operand>
</operation>
</expression>
```

XML-документ завжди має структуру дерева, у корені якого сам документ. Його частини, описувані вкладеними парами тегів, утворюють вузли. Таким чином, ребра дерева позначають "безпосереднє вкладення". Атрибути тегу вважають

листками, як і найглибше вкладені частини, що не мають у своєму складі інших частин. Бачимо, що документ має деревоподібну структуру.

На відміну від html, у XML одиночні (непарні) теги записують із косою рискою: <BR/>, а атрибути – у лапках. У XML у назвах тегів та атрибутів має значення регістр літер.

7.1.1. Формування XML-документа

Концептуально є два шляхи обробки XML-документа: послідовна обробка та робота з об'єктною моделлю документа.

У першому випадку звичайно використовується SAX (Simple API for XML, простий програмний інтерфейс для XML). Робота SAX полягає в читанні джерел даних (input source) XML аналізаторами (XML-reader) та генерації послідовності подій (events), які обробляються об'єктами-обробниками (handlers). SAX надає послідовний доступ до XML-документа.

В іншому разі аналізатор XML будує DOM (Document Object Model, об'єктна модель документа), пропонуючи для XML документа конкретну об'єктну модель. У рамках цієї моделі вузли DOM-дерева доступні для довільного доступу, а для переходів між вузлами передбачено ряд методів.

Можна використовувати обидва ці підходи для формування наведеного вище XML-документа.

За допомогою SAX документ сформується як показано у прикладі:

```
import sys
from xml.sax.saxutils import XMLGenerator
g = XMLGenerator(sys.stdout)
g.startDocument()
g.startElement("expression", {})
g.startElement("operation", {"type": "+"})
g.startElement("operand", {})
g.characters("2")
g.endElement("operand")
g.startElement("operand", {})
```

```
g.startElement("operation", {"type": "*"})
g.startElement("operand", {})
g.characters("3")
g.endElement("operand")
g.startElement("operand", {})
g.characters("4")
g.endElement("operand")
g.endElement("operation")
g.endElement("operand")
g.endElement("operation")
g.endElement("expression")
g.endDocument()
```

Побудова дерева об'єктної моделі документа може мати вигляд як у прикладі наведеному нижче:

```
from xml.dom import minidom
dom = minidom.Document()
e1 = dom.createElement("expression")
dom.appendChild(e1)
p1 = dom.createElement("operation")
p1.setAttribute('type', '+')
x1 = dom.createElement("operand")
x1.appendChild(dom.createTextNode("2"))
p1.appendChild(x1)
e1.appendChild(p1)
p2 = dom.createElement("operation")
p2.setAttribute('type', '*')
x2 = dom.createElement("operand")
x2.appendChild(dom.createTextNode("3"))
p2.appendChild(x2)
x3 = dom.createElement("operand")
x3.appendChild(dom.createTextNode("4"))
p2.appendChild(x3)
x4 = dom.createElement("operand")
x4.appendChild(p2)
p1.appendChild(x4)
print (dom.toprettyxml())
```

Легко помітити, що при використанні SAX команди на генерацію тегів та інших частин видаються послідовно, а ось побудову однієї й тієї самої DOM можна виконувати різними послідовностями команд щодо формування вузла та його з'єднання з іншими вузлами.

7.1.2. Аналіз XML-документа

Для роботи з готовим XML-документом потрібно скористатися XML-аналізаторами. Аналіз XML-документа зі створенням об'єкта класу Document відбувається всього в одному рядку, за допомогою функції *parse()*. В даному розділі розглядається стандартна бібліотека *xml.etree.ElementTree*. Розглянемо тестовий xml-файл, який має наступний вигляд:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank>68</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

Імпортуємо бібліотеку *ElementTree*.

```
>>> import xml.etree.ElementTree as etree
```

Бібліотека *ElementTree* - частина стандартної бібліотеки Python, в пакеті xml.etree.ElementTree.

```
>>> tree = etree.parse('examples/feed.xml')
```

Основною точкою входу в бібліотеку ElementTree є функція parse(), яка приймає ім'я файлу, або потоковий об'єкт. Функція парсить весь документ за раз.

```
>>> root = tree.getroot()
```

Функція *parse()* повертає об'єкт що являє собою ввесь документ. Це не кореневий елемент.

Також можливо парсити дані використовуючи строкову змінну за допомогою функції *fromstring()*.

```
>>> root = etree.fromstring('xmlstring')
```

Для того щоб отримати посилання на кореневий елемент - викличте метод *getroot()*.

```
>>> root
<Element 'data' at 0x02EB60F0>
```

Змінна *root має* властивості tag та attrib, які виводять атрибути та їх значення для кожного тегу:

```
for child in root:
        print(child.tag, child.attrib)

country {'name': 'Liechtenstein'}
```

country {'name': 'Singapore'}

country {'name': 'Panama'}

Для того, щоб отримати доступ для дочірніх елементів слід використовувати індекси:

```
>>> root[0][1].text
'2008'
```

Для того, щоб рекурсивно вивести всі дочірні елементи використовують функцію *iter():*

```
>>> for neighbor in root.iter('neighbor'): print(neighbor.attrib)

{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

findall() знаходить тільки елементи з тегом, які є прямими нащадками поточного елемента. *find()* знаходить перший дочірній елемент з певним тегом. Функція *text* отримує доступ до текстового вмісту елемента. *get()* отримує доступ до атрибутів елемента:

```
>>> for country in root.findall('country'):
        rank = country.find('rank').text
        name = country.get('name')
        print(name, rank)


Liechtenstein 1
Singapore 4
Panama 68
```

7.2. Формат CSV.

Використання CSV (comma-separated values – значення, розділені комами) є поширеним способом переміщення даних в додатки типу електронних таблиць і з таких додатків з використанням звичайного тексту. Вміст файлів CSV – це лише ряди строкових значень, розділених комами.

На перший погляд, синтаксичний аналіз в цьому випадку повинен виявитися вельми простим. Досить було б, наприклад, регулярно виконувати *str.split(',')*.

Розглянемо короткий приклад запису CSV-даних в файл і подальше зчитування з нього.

```
import csv

data = (
   (9,'Web Clients and Servers','base64, urllib'),
   (10,'Web Progranrning : CGI & WSGI', 'cgi, time, wsgiref' ),
   (13 , 'Web Services ' , 'urllib, twython'),
   )

print(' *** WRITING CSV DATA ')
f = open('bookdata.csv' , 'w', newline='')
writer = csv.writer(f)
for record in data:
   writer.writerow(record)
f.close( )

print( ' ** * REVIEW OF SAVED DATA ' )
f = open ( 'bookdata.csv' , 'r' )
reader = csv.reader(f)

for chap, title, modpkgs in reader :
   print ('Chapter %s : %r ( featuring % s )' % (chap, title, modpkgs ) )

f.close( )
```

Результат виконання даного коду наведено нижче:

****** WRITING CSV DATA***
***** * REVIEW OF SAVED DATA***
Chapter 9 : 'Web Clients and Servers' (featuring base64, urllib)
Chapter 10 : 'Web Progranrning : CGI & WSGI' (featuring cgi, time, wsgiref)
Chapter 13 : 'Web Services ' (featuring urllib, twython)

Спочатку ми імпортуємо модуль *csv*. Слідом за оператором імпорту розташований наш набір даних *data*, який складається з потрійних кортежів.

Функція *csv.writer()* отримує на вході об'єкт типу "*відкритий файл*" (або файлоподібний об'єкт) і повертає об'єкт, що виконує запис (записуючий пристрій). Пристрій запису використовує метод *writerow()*, який дозволяє записувати в певний файл рядки даних, що розділяються комами. Після виконання запису цей файл закривається.

Функція *csv.reader()* протилежна до *csv.writer()*: вона повертає ітеруємий об'єкт, який можна використовувати для зчитування і виконання синтаксичного аналізу кожного рядка CSV-даних. Подібно *csv.writer()*, функція *csv.reader()* також отримує на вході об'єкт типу "відкритий файл" і повертає об'єкт, що виконує читання (пристрій читання).

7.3. Формат JSON.

JSON (JavaScript Object Notation) - простий формат обміну даними, заснований на підмножині синтаксису JavaScript. Модуль json дозволяє кодувати і декодувати дані в зручному форматі.

Приклад кодування основних об'єктів Python в JSON:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
```

```
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Далі наведено приклад компактного кодування:

```
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'

>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Приклад декодування (парсинг) JSON-об’єктів в Python-структуру:

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
```

7.3.1. Основні методи

json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False) – серіалізує *obj* як форматований JSON потік в *fp*.

Якщо *skipkeys = True*, то ключі словника, які не входять до базового типу (*str, unicode, int, long, float, bool, None*) будуть проігноровані, замість того, щоб викликати виключення *TypeError*.

Якщо параметр *ensure_ascii* приймає значення *True*, всі не-ASCII символи у виводі будуть екрановані послідовностями \ *uXXXX*, і результатом буде рядок, що містить тільки ASCII символи. Якщо *ensure_ascii = False*, рядки запишуться як є.

Якщо параметр *check_circular* приймає значення *False*, то перевірка циклічних посилань буде пропущена, а такі посилання будуть викликати *OverflowError*.

Якщо *allow_nan = False*, при спробі серіалізовати значення дробних чисел, що

виходять за допустимі межі, буде викликатися *ValueError (nan, inf, -inf)* в суворій відповідності зі специфікацією JSON, замість того, щоб використовувати еквіваленти з *JavaScript (NaN, Infinity, -Infinity).*

Якщо *indent* є невід'ємним числом, то масиви і об'єкти в JSON будуть виводитися з цим рівнем вкладеності (доступу). Якщо рівень відступу 0, негативний або "", то замість цього просто використовуватимуться нові рядки. Значення за замовчуванням *None* відображає найбільш компактний вигляд.

При *sort_keys = True* словник буде відсортовано по ключам.

json.dumps (obj, skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, cls = None, indent = None, separators = None, default = None, sort_keys = False) - серіалізує obj в рядок JSON-формату. Аргументи мають те ж значення, що і для *dump()*.

Ключі в парах ключ/значення в JSON завжди є рядками. Коли словник конвертується в JSON, всі ключі словника перетворюються в рядки. В результаті цього, якщо словник спочатку перетворити в JSON, а потім назад в словник, то можна не отримати словник, ідентичний вихідному.

json.load (fp, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None) - перетворює JSON-дані в Python-структуру.

object_hook - опціональна функція, яка застосовується до результату декодування об'єкта (dict).

object_pairs_hook - опціональна функція, яка застосовується до результату декодування об'єкта з певною послідовністю пар ключ/значення.

Параметр *parse_float* застосовується для кожного значення JSON з плаваючою точкою. За замовчуванням, це еквівалентно float (num_str).

Параметр *parse_int*, застосовується для рядка JSON з числовим значенням. За замовчуванням еквівалентно int (num_str).

Параметр *parse_constant* застосовується для наступних рядків: "-Infinity", "Infinity", "NaN". Може бути використано для порушення винятків при виявленні помилкових чисел JSON.

Якщо не вдасться декодувати JSON, буде порушено виняток *ValueError*.

json.loads (s, encoding = None, cls = None, object_hook = None, parse_float = None, parse_int = None, parse_constant = None, object_pairs_hook = None) - декодує s (екземпляр str, що містить документ JSON) в об'єкт Python.

Інші аргументи аналогічні аргументам в load().

7.3.2. Класи модуля JSON

Клас *json.JSONDecoder(object_hook = None, parse_float = None, parse_int=None, parse_constant = None, strict = True, object_pairs_hook = None)* - простий декодер JSON. Виконує наступні перетворення при декодуванні:

Таблиця 7.1 – Декодування JSON-типів в Python-типи

JSON	**Python**
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Він також розуміє NaN, Infinity та Infinity як відповідні значення float, які знаходяться за межами специфікації JSON.Клас *json.JSONEncoder (skipkeys = False, ensure_ascii = True, check_circular = True, allow_nan = True, sort_keys = False, indent = None, separators = None, default = None)* – кодує структуру даних Python в JSON-об’єкт.Він підтримує наступні об'єкти і типи даних, які вказано в Табл.7.2:

Таблиця 7.2 – Кодування Python-типів в JSON

Python	**JSON**
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

ПРАКТИЧНА РОБОТА

1) Написати Python-скрипт, який повинен:
   - виконати пошук всіх CSV-файлів в поточному робочому каталозі;
   - прочитати повний вміст кожного файла;
   - записати зміст кожного файла без першого рядка (рядок заголовка) в CSV-файл з тим же ім'ям.

```
#! python3
# Видаляє заголовок з усіх файлів CSV у поточній
# директорії

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Цикл, що проходить кожний файл в поточній директорії
for csvFilename in os.listdir('.'):
   if not csvFilename.endswith('.csv'):
      continue # skip non-csv files

   print('Removing header from ' + csvFilename + '...')

   # Зчитує CSV-файл (пропускаючи перший рядок).
   csvRows = []
   csvFileObj = open(csvFilename)
   readerObj = csv.reader(csvFileObj)
   for row in readerObj:
      if readerObj.line_num == 1:
         continue# пропуска' перший рядок
      csvRows.append(row)
   csvFileObj.close()

   # Записує дані в CSV-файл.
   csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
newline='')
   csvWriter = csv.writer(csvFileObj)
   for row in csvRows:
```

```
    csvWriter.writerow(row)
  csvFileObj.close()
```

2) Написати Python-скрипт, який повинен:
   - зчитати з консолі назву населеного пункту (на англійській мові);
   - завантажує погодні дані в JSON-форматі з ресурсу openweathermap.org;
   - перетворює JSON-об'єкт в Python-структуру даних;
   - виводить прогноз погоди на сьогодні та на наступні два дні.

Перед виконанням завдання слід становити пакет *requests*, якщо він не встановлений:

a) для Windows: *pip3 install requests*

b) для Linux(Ubuntu): *pip3 install requests*

```
#! python3
# Виводить прогноз погоди для конкретного місця заданого з командної строки.

import json, requests, sys

# Визначає кількість переданих аргументів.
if len(sys.argv) < 2:
   print('Usage: quickWeather.py location')
   sys.exit()
location = ' '.join(sys.argv[1:])

# Завантажує JSON дані з OpenWeatherMap.org
url
='http://api.openweathermap.org/data/2.5/weather?q=%s&cnt=3&appid=707d738c92ca
835297381e002b64975a' % (location)
response = requests.get(url)
response.raise_for_status()

# Перетворює JSON дані в Python структуру.
weatherData = json.loads(response.text)
```

```
# Виводить дані про погоду.
w = weatherData['weather']
print('Current weather in %s:' % (location))
print(w[0]['main'], '-', w[0]['description'])
print()
print('Tomorrow:')
print(w[1]['main'], '-', w[1]['description'])
print()
print('Day after tomorrow:')
print(w[2]['main'], '-', w[2]['description'])
```

Самостійна робота

Виконати пункт 2 для формату даних XML.

Висновок

В даному розділі було розглянуто взаємодію мови програмування Python 3 з основними форматами даних. Пройшовши даний розділ студент зможе ознайомиться особливостями обробки даних в різних форматах (XML, CSV, JSON) в Python 3.

Питання для самоконтролю

1. Що таке XML? Наведіть приклад XML-документа.
2. Що таке DOM?
3. Яким чином можна провести аналіз XML-документа?
4. Дайте визначення CSV-формату.
5. Яким чином можна провести аналіз документа в CSV-форматі?
6. Що таке JSON? Наведіть приклад даних в форматі JSON. Які ви знаєте методи роботи з JSON-документами?

8. РОЗРОБКА WEB-ДОДАТКІВ.

Під веб-додатком (прикладною інтернет-програмою) розуміють програму, основний інтерфейс користувача якої працює в стандартному веб-браузері під керуванням html та XML документів. Для поліпшення якості інтерфейсу користувача часто використовують JavaScript, однак це дещо знижує універсальність інтерфейсу. Зазначимо, що інтерфейс можна побудувати на Java- або Flash-аплетах, однак, такі програми складно назвати веб-програмами, тому що Java або Flash можуть використовувати власні протоколи для спілкування із сервером, а не стандартний для www протокол HTTP.

При створенні веб-програм намагаються відокремити форму (зовнішній вигляд, стиль), зміст та логіку обробки даних. Сучасні технології побудови веб-сайтів дають можливість підійти досить близько до цього ідеалу. Проте, навіть не використовуючи багаторівневі програми, можна дотримуватися стилю, що дозволяє змінювати кожний із цих аспектів, не зачіпаючи (або майже не зачіпаючи) два інші.

8.1. CGI додатки

Додаток CGI (Common Gateway Interface, загальний шлюзовий інтерфейс) не на багато відрізняється від звичайної програми. Основні відмінності полягають в тому, що в цьому інтерфейсі засоби введення-виведення даних і взаємодія з користувачем реалізовані інакше, ніж в програмі, не призначеної для роботи в мережі. Після запуску сценарію CGI веб-сервер отримує дані, введені користувачем, але вони приходять від веб-клієнта, а не з локального комп'ютера або з файлу на диску. Цей процес називається запитом (request).

На відміну від звичайного програмного додатку, при виведенні даних CGI-додаток передає їх веб-клієнту, а не у вікно графічного інтерфейсу програми. Дані, що передаються сервером веб-клієнту, називаються відповіддю (response).

Нарешті, слід зазначити, що безпосередньо в самому сценарії взаємодія з користувачем не передбачено. Весь обмін даними відбувається між веб-клієнтом,

що діє від імені користувача та веб-сервером і додатком CGI.

Класичний шлях створення програм для веб-додатків – написання CGI-сценаріїв (іноді кажуть – скриптів). CGI – це стандарт, що регламентує взаємодію сервера із зовнішніми програмами. У випадку з www веб-сервер може направити запит на генерацію сторінки визначеному сценарію. Цей сценарій, отримавши на вхід дані від веб-сервера (той, у свою чергу, міг отримати їх від користувача), генерує готовий об'єкт (зображення, аудіодані, таблицю стилів тощо).

При виклику сценарію веб-сервер передає йому інформацію через стандартний потік введення, змінні оточення і, для *ISINDEX*, через аргументи командного рядка (вони доступні через *sys.argv*).

Два основні методи передавання даних із заповненої у браузері форми веб-сервера (і CGI-сценарію) – GET та POST. Залежно від методу дані передаються по-різному. У першому випадку вони кодуються та містяться прямо в URL, наприклад: http://host/cgi-bin/a.cgi?*a=1&b=3*. Сценарій отримує їх у змінній оточення з іменем *QUERY_STRING*. У випадку методу POST вони передаються на стандартний потік уведення. Для коректної роботи сценарії розміщують у призначеному для цього каталозі на веб-сервері (зазвичай він називається *cgi-bin*) або, якщо це дозволено конфігурацією сервера, у будь-якому місці серед документів html. Сценарій повинен буту виконуваним (у файловій системі). У системі Unix його можна встановити за допомогою команди chmod a+x.

Наступний найпростіший сценарій виводить значення зі словника *os.environ* та дозволяє побачити, що йому було передано:

```
#!/usr/bin/python
import os
print ("""Content-Type: text/plain
%s""" % os.environ)
```

За його допомогою можна побачити встановлені веб-сервером змінні оточення. CGI-сценарій, що видає веб-серверу файл, має заголовок, у якому містяться поля з метаінформацією (тип вмісту, час останнього відновлення

документа, кодування тощо).

Основні змінні оточення:
QUERY_STRING – рядок запиту.
REMOTE_ADDR – IP-адреса клієнта.
REMOTE_USER – ім'я клієнта (якщо він був ідентифікований).
SCRIPT_NAME – ім'я сценарію.
SCRIPT_FILENAME – ім'я файла зі сценарієм.
SERVER_NAME – ім'я сервера.
HTTP_USER_AGENT – назва браузера клієнта.
REQUEST_URI – рядок запиту (URI).
HTTP_ACCEPT_LANGUAGE – бажана мова документа.

8.1.1. Модуль cgi

У Python є підтримка CGI у вигляді модуля *cgi*. В модулі *cgi* передбачено основний клас *FieldStorage*, який виконує дану роботу. Цей клас зчитує всю необхідну інформацію про користувача, передану веб-клієнтом (через веб-сервер), тому копія цього класу повинна бути створена відразу після запуску CGI-сценарію у мові Python. Екземпляр вказаного класу включає об'єкт, що нагадує словник, який містить набір пар "ключ-значення". Ключовими є імена елементів вводу, переданих з заповненої форми. Значення містять відповідні дані.

Значення можуть представляти собою об'єкти одного з трьох типів. По перше, екземпляри класу *FieldStorage*. По-друге, екземпляри аналогічного класу *MiniFieldStorage*, який використовується в тих випадках, якщо не потрібна передача (upload) файлів або ведеться обробка даних форми, яка не складається з кількох частин. Екземпляри *MiniFieldStorage* містять тільки пари "ключ-значення", що складаються з імені та даних. По-третє, передані значення можуть являти собою список зазначених об'єктів. Такі списки формуються, якщо форма містить кілька елементів введення з однаковим ім'ям поля.

Для простих веб-форм зазвичай достатньо застосовувати лише екземпляри *MiniFieldStorage*.

8.1.2. Модуль cgitb

Для відладки CGI-сценарію можна використовувати модуль *cgit*b. При виникненні помилки цей модуль видасть html-сторінку, де вкаже місце виникнення виняткової ситуації. На початку налагоджуваного сценарію потрібно включити наступні рядки:

```
import cgitb
cgitb.enable()
```

Або, якщо не потрібно показувати помилки у браузері:

```
import cgitb
cgitb.enable(0, logdir="/tmp")
```

Тільки слід пам'ятати, що варто видалити ці рядки, коли сценарій буде налагоджено, тому що він видає частини коду сценарію. Цим можуть скористатися зловмисники, для того, щоб знайти вразливості у CGI-сценарії або підглянути паролі (якщо такі є у сценарії).

Наступний приклад демонструє деякі з можливостей даних модулів:

1) В поточному каталозі створіть Python-скрипт start.py, який буде запускати сервер на 8000 порту:

```
from http.server import HTTPServer, CGIHTTPRequestHandler
server_address = ("", 8000)
httpd = HTTPServer(server_address, CGIHTTPRequestHandler)
httpd.serve_forever()
```

2) Створіть в поточному каталозі HTML-форму під назвою forms.html, яка буде приймати дані від клієнта і посилати їх на обробку:

```
<!DOCTYPE html>
<html>
<head>
   <title>A simple form demonstration</title>
</head>
<body>
   <div style="text-align:center;">
```

```
    <h1>User login</h1>
    <form action="/cgi-bin/retrieval.py" method="get">
      username : <input type="text" name="username" style="text-align:center;">
      <br><br>
      password : <input type="password" name="password" style="text-align:center;">
      <br><br><br>
      <input type="submit" value="Submit">
    </form>
  </div>
</body>
</html>
```

3) В поточному каталозі створіть каталог cgi-bin в якому створіть Python-скрипт retrieval.py для обробки даних, що приходять з веб-сторінки:

```
#!/usr/bin/env python3

import cgi, cgitb
cgitb.enable()          ## дозволяє виправляти помилки з сценаріїв cgi в браузері
form = cgi.FieldStorage()

## зчитування даних з відповідних полів
first = form.getvalue('username')
last = form.getvalue('password')


print("Content-type:text/html\r\n\r\n")
print("<html>")
print("<head><title>User entered</title></head>")
print("<body>")
print("<h1>User has entered</h1>")
print("<b>Firstname : </b>" + first + "<br>")
print("<br><b>Lastname : </b>" + last + "<br>")
print("")
print("</div>")
```

```
print("</body>")
print("</html>")
```

4) Зайдіть в браузері за посиланням http://localhost:8000/forms.html , заповніть форму та підтвердіть введення

8.2. Стандарт WSGI

CGI-технологія не має перспектив подальшого розвитку, оскільки не забезпечує масштабування; процеси CGI створюються для обробки кожного окремого запиту, а потім знищуються (це можна порівняти з тим, як виконуються сценарії інтерпретатором Python). Якщо в веб-додаток будуть приходити тисячі запитів і у відповідь на це буде запускатися така ж кількість інтерпретаторів, то буде створене така навантаження, з яким не впорається жоден сервер. У зв'язку з цим отримав широке поширення наступні WSGI.

WSGI (Web Server Gateway Interface) — стандарт взаємодії між Python-програмою, яка виконується на стороні сервера, і самим веб-сервером, наприклад, Apache.

В Python існує велика кількість різного роду веб-фреймворків, інструментаріїв і бібліотек. У кожного з них власний метод встановлення та налаштування, вони часто написані так, що не можуть взаємодіяти між собою. Це може стати проблемою, оскільки вибір фреймворку може обмежити вибір веб-сервера і навпаки.

WSGI надає простий і універсальний інтерфейс для взаємодії між більшістю веб-серверів і веб-додатками чи фреймворками.

По стандарту WSGI, веб-застосунок має задовольняти наступним вимогам:

- має бути викличним (callable) об'єктом;
- приймати два параметри:
    - словник змінних оточення (environ);
    - обробник запиту(start_response)
- викликати обробник запиту з кодом HTTP-відповіді та HTTP-заголовками;
- повертати ітератор з тілом відповіді.

Простим прикладом WSGI-застосунку може служити така функція:

```
def simple_wsgi_app (environ, start_response) :
    status = ' 200 OK '
    headers = [( 'Content-type' , 'text/plain')]
    start_response ( status, headers)
    return [ 'Hello world !' ]
```

Додаток WSGI визначається як викликаний код, який завжди приймає такі параметри: словник змінних оточення сервера і ще один викликаний фрагмент коду, який ініціює підготовку відповіді з кодом статусу HTTP і заголовками HTTP для повернення клієнту. Цей код повинен повертати ітеруємий об'єкт, який становить корисні дані.

У наведеному вище додатку "Hello World" на основі WSGI ці змінні іменуються відповідно *environ* і *start_response()*

Змінна *environ* включає в себе змінні оточення, такі як *HTTP_HOS*T, *HTTP_USER_AGENT*, *SERVER_PROTOCOL* і т.д. Функція *start_response()*, яка повинна бути виконана в додатку, готує відповідь, що відправляється в кінцевому підсумку назад клієнту. Відповідь має включати код повернення HTTP (200, 300 і т.д.), а також відповіді HTTP- заголовки.

8.3. Введеня у фреймворк Django

Django – високорівневий відкритий Python-фреймворк для розробки веб-систем.

Сайт на Django будується з однієї або декількох частин, які рекомендується робити модульними..

Архітектура Django подібна на «Модель-Вид-Контролер» (MVC). Однак, те що називається «контролером» в класичній моделі MVC, в Django називається «вид», а те, що мало б бути «видом», називається «шаблон». Таким чином, MVC розробники Django називають MTV («Модель-Шаблон-Вид»).

Початкова розробка Django, як засобу для роботи новинних ресурсів, досить

сильно позначилася на його архітектурі: він надає ряд засобів, які допомагають у швидкій розробці веб-сайтів інформаційного характеру. Так, наприклад, розробнику не потрібно створювати контролери та сторінки для адміністративної частини сайту, в Django є вбудований модуль для керування вмістом, який можна включити в будь-який сайт, зроблений на Django, і який може керувати відразу декількома сайтами на одному сервері. Адміністративний модуль дозволяє створювати, змінювати і вилучати будь-які об'єкти наповнення сайту, протоколюючі всі дії, а також надає інтерфейс для управління користувачами і групами (з призначенням прав).

У дистрибутив Django також включені програми для системи коментарів, синдикації RSS і Atom, «статичних сторінок» (якими можна управляти без необхідності писати контролери та відображення), перенаправлення URL і т.д.

До основних можливостей фреймворку Django слід віднести:

1) Об'єктно-реляційне відображення (ORM)

Django підтримує парадигму ООП. Об'єкти БД в термінології Django іменуються «моделями». Фреймворк надає у розпорядження розробникові розвинутий прикладний програмний інтерфейс для високорівневого доступу до даних. В більшості випадків немає потреби писати SQL-запити.

Для прикладу, для проекту обліку учнів можна створити таку модель:

```
class Student(models.Model):
    name = models.CharField("Ім'я", max_length="100")
    surname = models.CharField("Прізвище", max_length="100")
    birth_date = models.DateField()
```

При виконанні синхронізації проекту з БД автоматично буде створена таблиця БД з полями, які відповідають полям (*properties*) моделі.

Вибірка всіх студентів:

```
students = Student.objects.all()
```

Вибірка з фільтром по прізвищу, по частині прізвища, по даті народження:

```
students = Student.objects.filter(surname="Іванов")
students = Student.objects.filter(surname__iexact="нов") # LIKE-фільтр
students = Student.objects.filter(birth_date__gte=datetime.date('1982', '4', '5'))
```

#старші за дану дату

2) Автоматична побудова інтерфейсу для адміністрування

Django автоматично створює CRUD (create read update delete – 4 базові функції управління даними: створення, зчитування, зміна і видалення)-інтерфейс ('адмінку').

3) Елегантні URL

Парсинг URL-адрес побудований на регулярних виразах. Розробник не обмежений у використанні певної схеми посилань.

4) Зручна система шаблонів

В Django є окрема мова для опису шаблонів. Вона є дуже простою для початківців. В ній присутні оператори циклу, умови, форматування даних.

5) Гнучка підсистема кешування даних

Django-проект може бути налаштований на роботу з Memcached чи будь-яким іншим фреймворком зберігання даних в оперативній пам’яті. Інструменти Django дозволяють кешувати SQL-вибірки, шаблони та їх частини і просто окремі змінні.

Детальну інформацію з використання фреймворка Django можна знайти за посиланням https://www.djangoproject.com/ .

ПРАКТИЧНА РОБОТА

1) За допомогою CGI створити веб-додаток, який оброблює дані та cockies веб-клієнта.

а) В поточному каталозі створити HTML-документ під назвою index.html з наступним кодом:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Обробка даних форми</title>
</head>
```

```
<body>
   <form action="/cgi-bin/form.py">
      <input type="text" name="TEXT_1">
      <input type="text" name="TEXT_2">
      <input type="submit">
   </form>
</body>
</html>
```

b) В каталозі cgi-bin створимо Python-скрипт для обробки даних форми під назвою form.py:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import cgi, unicodedata, html


form = cgi.FieldStorage()
text1 = form.getfirst("TEXT_1", "did not set")
text2 = form.getfirst("TEXT_2", "did not set")
text1 = html.escape(text1)
text2 = html.escape(text2)

print("Content-type: text/html\n")
print("""<!DOCTYPE HTML>
        <html>
        <head>
           <meta charset="utf-8">
           <title>Form data</title>
        </head>
        <body>""")

print("<h1>Form data!</h1>")
print("<p>TEXT_1: {}</p>".format(text1))
print("<p>TEXT_2: {}</p>".format(text2))

print("""</body>
        </html>""")
```

Тепер, перейшовши за посиланням http://localhost:8000/ та відіславши на сервер заповнені дані форми, можна побачити їх обробку за допомогою form.py.

c) В каталозі cgi-bin створимо Python-скрипт для обробки cookies користувача під назвою cookie.py:

```
#!/usr/bin/env python3
import os
import http.cookies

cookie = http.cookies.SimpleCookie(os.environ.get("HTTP_COOKIE"))
name = cookie.get("name")
if name is None:
   print("Set-cookie: name=value")
   print("Content-type: text/html\n")
   print("Cookies!!!")
else:
   print("Content-type: text/html\n")
   print("Cookies:")
   print(name.value)
```

Перейшовши за посиланням http://localhost:8000/cgi-bin/cookie.py перший раз – отримаємо результат: ***Cookies!!!***. Перезавантаживши дану веб-сторінку – результат буде наступним: ***Cookies: value***.

2) Створимо прототип веб-додатку "twitter".

a) В каталозі cgi-bin створимо Python-скрипт для обробки повідомлень користувача під назвою wall.py:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import cgi
import html
import http.cookies
import os, unicodedata
```

```
from _wall import Wall
wall = Wall()

cookie = http.cookies.SimpleCookie(os.environ.get("HTTP_COOKIE"))
session = cookie.get("session")
if session is not None:
    session = session.value
user = wall.find_cookie(session)

form = cgi.FieldStorage()
action = form.getfirst("action", "")

if action == "publish":
    text = form.getfirst("text", "")
    text = html.escape(text)
    if text and user is not None:
        wall.publish(user, text)
elif action == "login":
    login = form.getfirst("login", "")
    login = html.escape(login)
    password = form.getfirst("password", "")
    password = html.escape(password)
    if wall.find(login, password):
        cookie = wall.set_cookie(login)
        print('Set-cookie: session={}'.format(cookie))
    elif wall.find(login):
        pass
    else:
        wall.register(login, password)
        cookie = wall.set_cookie(login)
        print('Set-cookie: session={}'.format(cookie))

pattern = '''
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
```

```
<title> The wall</title>
</head>
<body>
    Login and registration form. If you enter a nonexistent name, a new user would
be registered.
    <form action="/cgi-bin/wall.py">
        Login: <input type="text" name="login">
        Password: <input type="password" name="password">
        <input type="hidden" name="action" value="login">
        <input type="submit">
    </form>

    {posts}

    {publish}
</body>
</html>
'''

if user is not None:
    pub = '''
    <form action="/cgi-bin/wall.py">
        <textarea name="text"></textarea>
        <input type="hidden" name="action" value="publish">
        <input type="submit">
    </form>
    '''
else:
    pub = ''

print('Content-type: text/html\n')

print(pattern.format(posts=wall.html_list(), publish=pub))
```

b) В каталозі cgi-bin створимо Python-скрипт для визначення функцій користувача користувача під назвою _wall.py:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
import json
import random
import time


class Wall:
    USERS = 'cgi-bin/users.json'
    WALL = 'cgi-bin/wall.json'
    COOKIES = 'cgi-bin/cookies.json'

    def __init__(self):
        """Створюємо початкові файли, якщо вони не створені"""
        try:
            with open(self.USERS, 'r', encoding='utf-8'):
                pass
        except FileNotFoundError:
            with open(self.USERS, 'w', encoding='utf-8') as f:
                json.dump({}, f)

        try:
            with open(self.WALL, 'r', encoding='utf-8'):
                pass
        except FileNotFoundError:
            with open(self.WALL, 'w', encoding='utf-8') as f:
                json.dump({"posts": []}, f)

        try:
            with open(self.COOKIES, 'r', encoding='utf-8'):
                pass
        except FileNotFoundError:
            with open(self.COOKIES, 'w', encoding='utf-8') as f:
                json.dump({}, f)

    def register(self, user, password):
        """Реєструє користувача. Повертає True при успішної реєстрації"""
        if self.find(user):
            return False  # Такий користувач існує
```

```
        with open(self.USERS, 'r', encoding='utf-8') as f:
            users = json.load(f)
        users[user] = password
        with open(self.USERS, 'w', encoding='utf-8') as f:
            json.dump(users, f)
        return True

    def set_cookie(self, user):
        """Записує куку в файл. Повертає створену куку."""
        with open(self.COOKIES, 'r', encoding='utf-8') as f:
            cookies = json.load(f)
        cookie = str(time.time()) + str(random.randrange(10**14))  # Генеруємо
унікальну куку для користувача
        cookies[cookie] = user
        with open(self.COOKIES, 'w', encoding='utf-8') as f:
            json.dump(cookies, f)
        return cookie

    def find_cookie(self, cookie):
        """По Куке знаходить своє ім'я користувача"""
        with open(self.COOKIES, 'r', encoding='utf-8') as f:
            cookies = json.load(f)
        return cookies.get(cookie)

    def find(self, user, password=None):
        """Шукає користувача по імені або по імені і паролю"""
        with open(self.USERS, 'r', encoding='utf-8') as f:
            users = json.load(f)
        if user in users and (password is None or password == users[user]):
            return True
        return False

    def publish(self, user, text):
        """публікує текст"""
        with open(self.WALL, 'r', encoding='utf-8') as f:
            wall = json.load(f)
        wall['posts'].append({'user': user, 'text': text})
        with open(self.WALL, 'w', encoding='utf-8') as f:
```

```
            json.dump(wall, f)

    def html_list(self):
        """Список постів для відображення на сторінці"""
        with open(self.WALL, 'r', encoding='utf-8') as f:
            wall = json.load(f)
        posts = []
        for post in wall['posts']:
            content = post['user'] + ' : ' + post['text']
            posts.append(content)
        return '<br>'.join(posts)
```

Перейшовши за посиланням http://localhost:8000/cgi-bin/wall.py можна побачити в дії створений веб-додаток.

Висновок

В даному розділі було розглянуто основні веб-додатки в мові програмування Python 3. Пройшовши даний розділ студент зможе ознайомиться з роботою веб-додатків стандарту CGI, WSGI та фреймворком Django в Python 3.

Питання для самоконтролю

1. Що таке веб-додаток?
2. Що таке CGI?
3. Які модулі підтримки CGI існують в Python 3?
4. Що таке WSGI?
5. Що таке Django?

9. МЕРЕЖНІ ДОДАТКИ НА PYTHON.

9.1. Робота із сокетами

Застосовувана в IP-мережах архітектура клієнт-сервер використовує IP-пакети для комунікації між клієнтом та сервером. Клієнт відправляє запит серверу, на який той відповідає. У випадку із TCP/IP між клієнтом та сервером встановлюється з'єднання (звичайно із двостороннім передаванням даних), а у випадку з UDP/IP – клієнт та сервер обмінюються пакетами (дейтаграмами) з негарантованою доставкою.

Кожний мережевий інтерфейс IP-мережі має унікальну в цій мережі адресу (IP-адресу). Спрощено можна вважати, що кожний комп'ютер у мережі Інтернет має власну IP-адресу. При цьому в рамках одного мережевого інтерфейсу може бути кілька мережевих портів. Для встановлення мережевого з'єднання програма клієнта повинна вибрати вільний порт та встановити з'єднання із серверною програмою, що слухає (listen) порт із визначеним номером на віддаленому мережевому інтерфейсі. Пара IP-адреса та порт характеризують сокет (гніздо) – початкову (та кінцеву) точку мережевої комунікації. Для створення з'єднання TCP/IP необхідно два сокети: один на локальній машині, а інший – на віддаленій. Таким чином, кожне мережеве з'єднання має IP-адресу та порт на локальній машині, а також IP-адресу та порт на віддаленій машині.

Модуль *socket* забезпечує можливість працювати із сокетами в Python. Сокети використовують транспортний рівень згідно з семирівневою моделлю OSI (Open Systems Interconnection, взаємодія відкритих систем), тобто належать до нижчого рівня, ніж більшість описуваних у цьому підрозділі протоколів.

Наведемо рівні моделі OSI.

Фізичний – потік бітів, переданих по фізичній лінії. Визначає параметри фізичної лінії.

Канальний (Ethernet, PPP, ATM тощо) кодує та декодує дані у вигляді потоку бітів, справляючись із помилками, що виникають на фізичному рівні в межах фізично єдиної мережі.

Мережевий (IP) маршрутизує інформаційні пакети від вузла до вузла.

Транспортний (TCP, UDP тощо) забезпечує прозоре передавання даних між двома точками з'єднання.

Сеансовий керує сеансом з'єднання між вузлами мережі. Починає, координує та завершує з'єднання.

Представлення забезпечує незалежність даних від форми їхнього подання шляхом перетворення форматів. На цьому рівні може виконуватися прозоре (із погляду вищого рівня) шифрування та дешифрування даних.

Прикладний (HTTP, FTP, SMTP, NNTP, POP3, IMAP тощо) підтримує конкретні мережеві застосування. Протокол залежить від типу сервісу.

Кожний сокет належить до одного з комунікаційних доменів. Модуль *socket* підтримує домени UNIX та Internet. Кожен домен стосується свого набору протоколів та адресації. Цей підрозділ порушуватиме тільки питання домену Internet, а саме протоколи TCP/IP та UDP/IP, тому для вказання комунікаційного домену при створенні сокета вказуватиметься константа *socket.AF_INET*.

Як приклад розглянемо найпростішу клієнт-серверну пару.

Сервер прийматиме рядок та відповідатиме клієнту. Мережевий пристрій іноді називають хостом (host), тому цей термін буде вживатися стосовно комп'ютера, на якому працює мережева програма.

Серверна частина програми:

```
import socket, string

HOST = "127.0.0.1" # localhost
PORT = 33333
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind((HOST, PORT))

while True:
   print ("Слухаю порт 33333")
   srv.listen(1)
   sock, addr = srv.accept()
   while True:
```

```
        pal = sock.recv(1024)
        if not pal:
            break
        print ("Отримано від %s:%s:" % addr, pal)
        lap = 'The message has been processed'
        print ("Відправлено %s:%s:" % addr, lap)
        sock.send(b'ok, request')
    sock.close()
```

Клієнтська частина програми:

```
import socket, os
HOST = "127.0.0.1" # віддалений комп'ютер (localhost)
PORT = 33333# порт на віддаленому комп'ютері
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))
sock.send(b'Hello world!!!')
result = sock.recv(1024)
sock.close()
print ("Отримано:", result)
```

Насамперед, потрібно запустити сервер. Сервер відкриває сокет на локальній машині на порту 33333 та за адресою 127.0.0.1. Після цього він слухає (*listen()*) порт. Коли на порту з'являються дані, приймається (*accept()*) вхідне з'єднання. Метод *accept()* повертає пару – Socket-об'єкт та адресу віддаленого комп'ютера, що встановлює з'єднання (пара – IP-адреса, порт на віддаленій машині). Після цього можна застосовувати методи *recv()* та *send()* для спілкування із клієнтом. У *recv()* указується кількість байтів у черговій порції. Від клієнта може прийти й менша кількість даних.

Код програми-клієнта досить очевидний. Метод *connect()* установлює з'єднання з віддаленим хостом (у наведеному прикладі він розташований на тій самій машині). Дані передаються методом *send()* та приймаються методом recv() – аналогічно тому, як це відбувається на сервері.

Модуль *socket* має кілька допоміжних функцій. Зокрема, функції для роботи із системою доменних імен (DNS):

```
>>> import socket
>>> socket.gethostbyname('www.ukr.net')
'212.42.76.253'
>>> socket.gethostbyaddr('212.42.76.253')
('srv253.fwdcdn.com', [], ['212.42.76.253'])
>>> socket.gethostname()
'test-host'
```

Функція *socket.getservbyname()*, яка дозволяє перетворювати найменування інтернет-сервісів у загальноприйняті номери портів:

```
import socket, string

for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':
    print (socket.getservbyname(srv, 'tcp'), srv)

80 http
21 ftp
143 imap
110 pop3
25 smtp
```

9.2. Модуль smtplib

Повідомлення електронної пошти в Інтернеті передаються від клієнта до сервера та між серверами в основному за протоколом SMTP (Simple Mail Transfer Protocol, простий протокол передавання пошти). Протоколи SMTP та ESMTP (розширений варіант SMTP) описано в RFC 821 та RFC 1869. Для роботи з SMTP у стандартній бібліотеці модулів є модуль *smtplib*. Для того, щоб почати SMTP-з'єднання із сервером електронної пошти, необхідно на початку створити об'єкт для керування SMTP-сесією за допомогою конструктора класу SMTP:

```
smtplib.SMTP([host[, port]])
```

Параметри *host* та *port* визначають адресу та порт SMTP-сервера, через який відправлятиметься пошта. За замовчуванням , port = 25. Якщо host задано, конструктор сам установить з'єднання, інакше доведеться окремо викликати метод *connect()*. Екземпляри класу SMTP мають методи для всіх розповсюджених команд SMTP-протоколу, але для відправлення пошти достатньо виклику конструктора та методів *sendmail()* та *quit()*:

```
# -*- coding: cp1251 -*-
from smtplib import SMTP
fromaddr = "timurdov@ukr.net" # Від кого
toaddr = "antoniy_kovalsky@ukr.net"# Кому
message = """From: Student <%(fromaddr)s>
To: Lecturer <%(toaddr)s>
Subject: From Python course student
MIME-Version: 1.0
Content-Type: text/plain; charset=Windows-1251
Content-Transfer-Encoding: 8bit
Добрий день! Я вивчаю курс з мови Python і
відправляю лист його авторові.
"""
connect = SMTP('mail.ukr.net')
connect.set_debuglevel(1)
connect.sendmail(fromaddr, toaddr, message % vars())
connect.quit()
```

Зазначимо, що *toaddr* у повідомленні (у полі To) та при відправленні можуть не збігатися тому, що параметри (одержувач та відправник) у ході SMTP-сесії передаються командами SMTP-протоколу. При запуску зазначеного вище прикладу на екрані повинна з'явиться налагоджувальна інформація (адже рівень налагодження вказаний рівним 1)

Під час однієї SMTP-сесії можна надіслати відразу кілька листів поспіль, якщо не викликати quit().

Команди SMTP можна подавати й окремо: для цього в об'єкта-з'єднання є методи *(helo(), ehlo(), expn(), help(), mail(), rcpt(), vrfy(), send(), noop(), data()*), що відповідають однойменним командам SMTP-протоколу.

При роботі з класом smtplib.SMTP можуть генеруватися різні виняткові ситуації. Призначення деяких із них наведено нижче:

- smtplib.SMTPException – базовий клас для всіх виняткових ситуацій модуля;
- smtplib.SMTPServerDisconnected – сервер неочікувано перервав зв'язок (або зв'язок із сервером не було встановлено);
- smtplib.SMTPResponseException – базовий клас для всіх виняткових ситуацій, які мають код відповіді SMTP-сервера;
- smtplib.SMTPSenderRefused – відправника відкинуто;
- smtplib.SMTPRecipientsRefused – сервер відкинув усіх отримувачів;
- smtplib.SMTPDataError – сервер відповів невідомим кодом на повідомлення;
- smtplib.SMTPConnectError – помилка встановлення з'єднання;
- smtplib.SMTPHeloError – сервер не відповів правильно на команду HELO або відкинув її.

9.3. Модуль poplib

Ще один протокол – POP3 (Post Office Protocol, поштовий протокол) служить для прийому пошти з поштової скриньки на сервері (протокол визначено у RFC 1725).

Для роботи з поштовим сервером потрібно встановити з ним з'єднання і, подібно до розглянутого вище прикладу, за допомогою SMTP-команд отримати необхідні повідомлення. Об'єкт з'єднання POP3 можна встановити за допомогою конструктора класу POP3 із модуля *poplib*:

poplib.POP3(host[, port])

де *host* – адреса POP3-сервера, *port* – порт на сервері (за замовчуванням 110), *pop_obj* – об'єкт для керування сеансом роботи з POP3-сервером.

Наступний приклад демонструє основні методи для роботи з POP3-з'єднанням (для того, щоб приклад спрацював коректно, необхідно внести реальні облікові дані користувача на сервері):

```
import poplib, email
# Облікові дані користувача:
SERVER = "pop.server.com"
USERNAME = "user"
USERPASSWORD = "secretword"
p = poplib.POP3(SERVER)
print (p.getwelcome())
# етап ідентифікації
print (p.user(USERNAME))
print (p.pass_(USERPASSWORD))
# етап транзакцій
response, lst, octets = p.list()
print (response)
for msgnum, msgsize in [i.split() for i in lst]:
   print ("Повідомлення %(msgnum)s має довжину% (msgsize)s" % vars())
   print ("UIDL =", p.uidl(int(msgnum)).split()[2])
   if int(msgsize) > 32000:
      (resp, lines, octets) = p.top(msgnum, 0)
   else:
      (resp, lines, octets) = p.retr(msgnum)
   msgtxt = "\n".join(lines)+"\n\n"
   msg = email.message_from_string(msgtxt)
   print ("* Від: %(from)s\n* Кому: %(to)s\n* Тема:%(subject)s\n" % msg)
   # msg містить заголовки повідомлення або все повідомлення (якщо воно невелике)
   # етап відновлення

print (p.quit())
```

Ці й інші методи екземплярів класу POP3 описано в табл. 9.1.

Таблиця 9.1 – екземпляри класу POP3

Метод	Команда POP3	Опис
getwelcome()		Отримує рядок s із вітанням POP3-сервера
user(name)	**USER name**	Надсилає команду USER, вказуючи ім'я користувача name. Повертає рядок із відповіддю сервера
pass_(pwd)	**PASS pwd**	Надсилає пароль користувача в команду PASS. Після цієї команди і до виконання команди QUIT поштова скринька блокується
apop(user, secret)	**APOP user secret**	Виконує ідентифікацію на сервері за APOP
rpop(user)	**RPOP user**	Здійснює ідентифікацію за методом RPOP
stat()	**STAT**	Повертає кортеж з інформацією про поштову скриньку. У ньому m – кількість повідомлень, l – розмір поштової скриньки в байтах
list([num])	**LIST [num]**	Повертає список повідомлень у форматі (resp, ['num octets', ...]), якщо не вказано num, і "+OK num octets", якщо вказано. Список lst складається з рядків у форматі "num octets"
retr(num)	**RETR num**	Завантажує з сервера повідомлення з номером num та повертає кортеж із відповіддю сервера (resp, lst, octets)
dele(num)	**DELE num**	Видаляє повідомлення з номером num
rset()	**RSET**	Скасовує позначки видалення повідомлень
noop()	**NOOP**	Не виконує функцій (підтримує з'єднання)

Таблиця 9.1 (продовження) – екземпляри класу POP3

quit()	**QUIT**	Від'єднує від сервера. Сервер виконує всі необхідні зміни (видаляє повідомлення) та знімає блокування поштової скриньки
top(num, lines)	**TOP num lines**	Команда аналогічна до RETR, але завантажує тільки заголовок та lines рядків тіла повідомлення. Повертає кортеж (resp, lst, octets)
uidl([num])	**UIDL [num]**	Виконує скорочення від "unique-id listing" (список унікальних ідентифікаторів повідомлень). Формат результату: (resp, lst, octets), якщо num не вказано, і "+OK num uniqid", якщо вказано. Список lst складається з рядків вигляду "+OK num uniqid"

У цій таблиці *num* означає номер повідомлення (він не змінюється протягом усієї сесії); *resp* – відповідь сервера, повертається для будь-якої команди, починається з "+OK " для успішних операцій (при невдачі генерується виняткова ситуація виняткові ситуації *poplib.proto_error*). Параметр *octets* визначає кількість байтів у прийнятих даних, *uniqid* – ідентифікатор повідомлення, генерується сервером.

Робота з POP3-сервером складається з трьох фаз: ідентифікації, транзакцій та відновлення. На етапі ідентифікації відразу після створення POP3-об'єкта дозволено тільки команди USER, PASS (іноді APOP та RPOP). Після ідентифікації сервер отримує інформацію про користувача і наступає етап транзакцій. Тут доступні інші команди. Етап відновлення викликається командою QUIT, після якої POP3-сервер обновляє поштову скриньку користувача відповідно до поданих команд, а саме – видаляє позначені для видалення повідомлення.

9.4. Модулі для клієнта www

Стандартні засоби мови Python дозволяють отримувати з програми доступ до об'єктів www як у простих випадках, так і за складних обставин, зокрема при необхідності передавати дані форми, ідентифікації, доступу через проксі тощо.

Зазначимо, що при роботі з www використовується здебільшого протокол HTTP, однак www охоплює не тільки HTTP, але й багато інших схем (FTP, HTTPS тощо). Використовувана схема зазвичай вказується на самому початку URL.

9.4.1. Функції для завантаження мережевих об'єктів

Простий випадок отримання веб-об'єкта за відомим URL наведено в такому прикладі:

```
import urllib.request as req
doc = req.urlopen("http://python.org").read()
print (doc[:40])
```

Функція *urllib.urlopen()* створює файлоподібний об'єкт, який можна читати методом *read()*. Інші методи цього об'єкта: *readline(), readlines(), fileno(), close()* працюють як і у звичайного файла, а також є метод *info()*, що повертає message-об'єкт, відповідний отриманим із сервера даним. Цей об'єкт можна використовувати для одержання додаткової інформації:

```
import urllib.request as req
f = req.urlopen("http://python.org")
print (f.info())
```

Результатом буде:

```
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
x-xss-protection: 1; mode=block
X-Clacks-Overhead: GNU Terry Pratchett
Via: 1.1 varnish
Fastly-Debug-Digest:
a63ab819df3b185a89db37a59e39f0dd85cf8ee71f54bbb42fae41670ae56fd2
```

```
Content-Length: 48844
Accept-Ranges: bytes
Date: Sun, 11 Mar 2018 14:02:13 GMT
Via: 1.1 varnish
Age: 1458
Connection: close
X-Served-By: cache-iad2120-IAD, cache-hhn1531-HHN
X-Cache: HIT, HIT
X-Cache-Hits: 3, 16
X-Timer: S1520776934.559399,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

За допомогою функції *urllib.urlopen()* можна робити і складніші речі, наприклад, передавати веб-серверу дані форми. Як відомо, дані заповненої веб-форми можуть бути передані на вебсервер із використанням методу GET або методу POST. Метод GET пов'язаний із кодуванням усіх переданих параметрів після знака "?" в URL, а при методі POST дані передаються в тілі HTTP-запиту. Обидва варіанти передавання наведено нижче:

```
import string, urllib.parse  as par
import urllib.request as req
data = {"q": "Python"}
enc_data = par.urlencode(data)
# метод GET
f = req.urlopen("http://searchengine.com/search" + "?" + enc_data)
print (f.read())
# метод POST
f = req.urlopen("http://searchengine.com/search", str.encode(enc_data))
print (f.read())
```

Функція *urlretrieve()* дозволяє записати вказаний URL мережевий об'єкт у файл. Вона має такі параметри:

urllib.urlretrieve(url[, filename[, reporthook[, data]]]).

Тут url – URL мережевого об'єкта; filename – ім'я локального файла для розміщення об'єкта; reporthook – функція, що буде викликатися для повідомлення

про стан завантаження; data – дані для методу POST (якщо він використовується). Функція повертає кортеж (filepath, headers), де filepath – ім'я локального файла, у який завантажено об'єкт; headers – результат методу info() для об'єкта, який було повернуто urlopen().

Для забезпечення інтерактивності функція urllib.urlretrieve() викликає час від часу функцію, указану в reporthook. Цій функції передаються три аргументи: кількість прийнятих блоків, розмір блоку та загальний розмір прийнятого об'єкта в байтах (якщо він невідомий, цей параметр дорівнює –1).

9.5. XML-RPC-сервер

Дотепер високорівневі протоколи розглядалися з погляду клієнта. Не менш просто створювати на Python і їхні серверні частини. Для ілюстрації того, як розробити програму на Python, яка реалізує сервер, обрано протокол XML-RPC. Незважаючи на свою назву, кінцевому користувачу не обов'язково знати XML, тому що її використання приховано від нього. Скорочення RPC (Remote Procedure Call, виклик віддаленої процедури) пояснює суть справи: за допомогою XML-RPC можна викликати процедури на віддаленому хості. Причому за допомогою XML-RPC можна абстрагуватися від конкретної мови програмування за рахунок використання загальноприйнятих типів даних (рядки, числа, логічні значення тощо). У мові Python виклик віддаленої функції за синтаксисом нічим не відрізняється від виклику звичайної функції:

```
import xmlrpc.client as cl
# Встановити з'єднання
req = cl.ServerProxy("http://localhost:8000")
try:
# Викликати віддалену функцію
   print (req.add(1, 3))
except cl.ProtocolError as err:
   print ("ERROR:: %s", err)
```

А ось як виглядає XML-RPC-сервер (для того, щоб перевірити роботу наведеного вище прикладу, необхідно спочатку запустити сервер):

```
from xmlrpc.server import SimpleXMLRPCServer

srv = SimpleXMLRPCServer(("localhost", 8000))# Запустити сервер
srv.register_function(pow)# Зареєструвати функцію
srv.register_function(lambda x,y: x+y, 'add')# І ще одну
srv.serve_forever()# Обслуговувати запити
```

За допомогою XML-RPC (а цей протокол досить "легкий" порівняно з іншими протоколами такого призначення) програми можуть спілкуватися одна з одною на зрозумілій їм мові виклику функцій із параметрами основних загальноприйнятих типів та таких самих значень, що повертаються. Перевагою в Python є зручний синтаксис виклику віддалених функцій.

ПРАКТИЧНА РОБОТА

1) На основі XML-RPC розробити клієнт-серверний додаток, який буде отримувати серверний час:

а) Серверна частина:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
```

```
server.serve_forever()
```

b) Клієнтська частина:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

2) На основі XML-RPC розробити клієнт-серверний додаток, який буде у запитах на сервер використовувати об'єкт MultiCall:

a) Серверна частина:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
   return x + y

def subtract(x, y):
   return x - y

def multiply(x, y):
   return x * y

def divide(x, y):
   return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
```

```
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

b) Клієнтська частина:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

3) Використовуючи PUT-запит, відіслати будь-які дані на http://localhost:8080 :

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Висновок

В даному розділі було розглянуто основні мережеві бібліотеки в мові програмування Python 3. Пройшовши даний розділ студент зможе ознайомиться з роботою модулів підтримки мережевих протоколів в Python 3, а також зможе написати простий клієнт/серверний додаток.

Питання для самоконтролю

1. Що таке сокет?
2. Що таке SMTP? Який модуль підтримки SMTP є в Python 3?
3. Що таке POP3? Який модуль підтримки POP3 є в Python 3?
4. Наведіть приклади бібліотек для роботи з веб-мережею в Python 3.
5. Що таке RPC?
6. Реалізуйте простійший клієнт/серверний додаток на основі RPC.

ВИСНОВКИ

В даному методичному посібнику була викладена основна теоретична інформація щодо мови програмування Python версії 3 (3.5 та вище), а також її практичне застосування.

Пройшовши даний курс студент буде:

1) знати:

- основні алгоритмічні конструкції та типи даних мови Python;
- основні стандартні бібліотеки;
- методи опису функцій на мові програмування Python;
- об’єктно-орієнтований підхід для вирішення прикладних задач програмування;
- чисельні алгоритми та матричні обчислення;
- основи роботи з текстом мови Python;
- основні модулі мови Python для роботи з різними форматами даних;
- структуру, основні можливості веб-фреймворків;
- основні вбудовані Python-модулі для роботи в мережі Інтернет.

2) вміти:

- складати програми мовою Python;
- підключати стандартні бібліотеки в Python-програму;
- функціонально описувати алгоритмічні конструкції на мові Python;
- складати застосовувати об’єктно-орієнтований підхід при розв’язуванні типових задач;
- використовувати математичні шаблони при розв’язуванні складних задач;
- розділяти та об’єднувати текст;
- перетворювати дані з одного формату в інший;
- розроблювати веб-додатки за допомогою веб–фреймворків на мові Python;

- робити запити на віддлений сервер за допомогою REST-FUL інтерфейсів.

З огляду на зазначене вище, містить актуальну на сьогоднішній день інформацію і цілком підходить для підготовки студентів ВНЗ по вивченнню мови програмування Python 3.

СПИСОК ЛІТЕРАТУРИ

1. Лутц М. Изучаем Python. – СПб. : Символ-плюс, 2015 -1272с.
2. Свейгарт Э. Автоматизация рутинных задач с помощью Python. Практическое руководство для начинающих. – К.: Вильямс, 2017 – 584с.
3. Мэтиз Э. Изучаем Python. Программирование игр, визуализация данных, веб-приложения. – СПб. : Питер, 2017 -492с.
4. Чан У. Дж. Библиотека професионала. Python. Создание приложений. 3 изд. . – К.: Вильямс, 2015 – 808с.
5. Прохоренок Н. Python 3 и PyQt. Разработка приложений. – СПб. : БХВ-Петербург, 2015 -704с.
6. https://www.python.org/
7. https://pythonworld.ru/